The ML Programming Guide

Programming Object-Oriented Image Processing with the MeVis Library

The ML Programming Guide

Published 2024-09-10

Table of Contents

About This Document 6
1. What This Document Contains 6
2. What You Should Know Before Reading This Document 8
3. Suggestions for Further Reading 8
4. Conventions Used in This Document 8
5. Quick Start 9
1. Conceptual Overview 10
1.1. Overview 11
1.2. Principles 11
1.3. ML Classes - Overview 13
1.3.1. Classes for Module Development 13
1.3.2. Administrative Classes 13
1.3.3. Image Classes 14
1.3.4. Helper Classes 15
1.3.5. APIs and Classes for Interfaces and Voxel Type Extensions 16
1.3.6. Component Groups 16
1.3.7. The ML Module Database 17
2. Detailed Class Overview and Usage 19
2.1. Classes for Module Development 21
2.1.1. `Module` 21
2.1.2. `Field` 21
2.1.3. `FieldContainer` 25
2.1.4. Image Classes for Module Development 27
2.2. Administrative Classes 27
2.2.1. `Host` 27
2.2.2. `Memory` 30
2.2.3. `Base` 32
2.2.4. The Runtime Type System 32
2.2.5. Debugging and Error Handling Support 34
2.3. Image Classes 34
2.3.1. `ImageProperties` 34
2.3.2. `MedicalImageProperties` 34
2.3.3. `ImagePropertyExtension` 35
2.3.4. `PagedImage` 37
2.3.5. `SubImage`/`TSubImage` 37
2.3.6. `BitImage` 38
2.3.7. `VirtualVolume` 39
2.3.8. `MemoryImage` 42
2.4. Helper Classes 44
2.4.1. `ImageVector`, ImageVector 44
2.4.2. `SubImageBox` 45
2.5. APIs and Classes for Interfaces and Voxel Type Extensions 46
2.5.1. How Applications and the ML Work 46
2.5.2. The C-API 46
2.5.3. Registering and Using Self-Defined Data Types 46
2.6. Tools 47
2.6.1. `MLLinearAlgebra`(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector) 47
2.6.2. `MLUtilities` 47
2.6.3. Other Classes 48
2.6.4. `MLBase` 50
2.6.5. `MLKernel` 50
2.6.6. `MLTools` 50
2.6.7. `MLDiagnosis` 51
2.6.8. `MLImageFormat` 52

2.6.9. `MLDataCompressors` 53
2.7. Registered Data Types 57
2.8. ML Data Types 57
2.8.1. Voxel Types and Their Enumerators 57
2.8.2. Index, Size and Offset Types 58
3. Deriving Your Own Module from Module 60
3.1. Deriving from `Module` 61
3.1.1. Basics 61
3.1.2. Implementing the Constructor 63
3.1.3. Module Persistence and Overloading activateAttachments() 65
3.1.4. Implementing handleNotification() 66
3.1.5. Using `TypedCalculateOutputImageHandler` 66
3.1.6. Implementing calculateOutputImageProperties() 67
3.1.7. Implementing calculateInputSubImageBox() 68
3.1.8. Changes to calcInSubImageProps() 69
3.1.9. Implementing calculateOutputSubImage() 69
3.1.10. Handling Disconnected or Invalid Inputs by Overloading handleInput() 74
3.1.11. Configuring Image Processing Behavior of the Module 75
3.1.12. Explicit Image Data Requests from Module Inputs 80
3.1.13. Getting Single Voxel Values from Module Inputs 81
3.1.14. Interrupting Page-Based Image Processing and Handling Errors 82
3.1.15. Testing for Interruptions During Calculations 82
3.1.16. Adapting Page Extents 83
3.1.17. Processing Input Images Sequentially 84
3.1.18. Traps and Pitfalls in Classes Derived from Module 86
4. Image Processing Concepts 89
4.1. Page Calculation in the ML 90
4.2. Page-Based Approaches 90
4.2.1. Page-Based Concept 90
4.2.2. Voxel-Based Concept 91
4.2.3. Slice-Based Concept 93
4.2.4. Kernel-Based Concept 95
4.3. Concepts for Partially Global Image Processing 95
4.3.1. Random Access Concept (Tile Requesting) 95
4.3.2. Sequential Image Processing Concept 96
4.3.3. `VirtualVolume` Concept 97
4.4. Global Image Processing Concepts 97
4.4.1. Temporary Global Concept 97
4.4.2. Global Image Processing Concept 98
4.4.3. `BitImage` Concept 99
4.4.4. `MemoryImage` Concept 99
4.5. Miscellaneous Modules 99
5. Debugging and Error Handling 101
5.1. Printing Debug Information 102
5.2. Handling Errors 105
5.3. Registering Error Handlers 106
5.4. The Class `ErrorOutput` and Configuring Message Outputs 107
5.5. Tracing, Exception Handling and Checked Object Construction/Destruction 109
6. The C-API 113
6.1. The C-API 113
6.2. mlInitSystemML.h 113
6.3. mlAPI.h 113
6.4. mlDataTypes.h 114
6.5. mlTypeDefs.h 114
6.6. C-Example using the C-API 115
7. Registered Voxel Data Types 118
7.1. Overview of Registered Voxel Data Types 119
7.1.1. Registered Voxel Data Types 119

7.1.2. About Standard, Default and Registered Voxel Types 120
7.2. Implementing Image Processing on extended Voxel Data Types 122
7.2.1. Important Functions For Voxel Types 123
7.2.2. The Basic Concept of Calculating the Output SubImage 124
7.2.3. Examples with Registered Voxel Types 124
7.2.4. Compile and Runtime Decisions on Standard and Registered Voxel Types 130
7.2.5. Handling Generalized Registered Voxel Types as Module Parameters 131
7.3. Limitations of Registered Data Types 133
7.4. Traps and Pitfalls When Using Registered Voxel Types 134
7.5. Advanced Issues on Registered Voxel Types 135
7.5.1. About the Difference Between Scalar, Extended and Registered Voxel Types... 135
7.5.2. Getting and Managing Metadata About Registered Voxel Types 137
7.5.3. Reducing Generated Code and Compile Times 144
7.5.4. Configuration of Supported Voxel Types 146
7.5.5. Implementing a New Voxel Data Type by Deriving from MLTypeInfos 147
8. Base Objects 153
8.1. `Base` Objects 154
8.2. Composing, Storing and Retrieving `Base` Objects 154
8.3. Creating Trees from `Base` Objects Using TreeNodes 154
8.4. Writing/Reading `Base` Objects to/from AbstractPersistenceStream 156
9. Unicode Support 159
9.1. Unicode Support 159
10. File System Support 161
10.1. File System 161
A. Basics about ML Programming and Projects 164
A.1. Creating an ML Project by Using MeVisLab 165
A.2. Programming Examples 165
A.3. Exporting Library Symbols 166
A.4. General Rules for ML Programming 167
A.5. How to Document an ML Module 168
A.6. Updating from Older ML Versions 169
A.7. Version Control 170
B. Optimizing Image Processing 173
B.1. Optimizing Module Code 173
B.2. Optimizing Data Flow in Module Networks 174
C. Handling Memory Problems 176
D. Messages and Errors 179
D.1. ML Error Codes 179
E. Improving Quality of ML-Based Software 186
Glossary 187

About This Document

This document describes nature, contents, usage and ways to enlarge the *MeVis Image Processing Library* (ML), often also called *MeVis Library*.

1. What This Document Contains

Chapter 1, *Conceptual Overview* provides information on the ML itself, its purpose, and its components.

Chapter 2, *Detailed Class Overview and Usage* gives a detailed survey of the most important ML classes and discusses their purpose and usage.

Chapter 3, *Deriving Your Own Module from Module* explains in detail how to implement your own image processing module by deriving a new class from the class `Module`.

Chapter 4, *Image Processing Concepts* addresses some concepts to find optimal implementation strategies for different algorithm types.

Chapter 5, *Debugging and Error Handling* gives a detailed introduction into the error and message handling system, as well as in logging and exception handling functionality of the ML.

Chapter 6, *The C-API* is an introduction to the C programming interface of the ML which can be used by applications and other programming languages.

Chapter 7, *Registered Voxel Data Types* shows how registered voxel data types work, how they can be used, implemented and registered in the ML.

Chapter 8, *Base Objects* describes how to (re)store self defined class objects with ML concepts.

Chapter 9, *Unicode Support* discusses how international/unicoded characters are handled by the ML.

Chapter 10, *File System Support* describes how files can be managed platform independently with international/unicoded file names.

Appendix A, *Basics about ML Programming and Projects* discusses some programming and system requirements needed to implement ML modules.

Section A.1, “Creating an ML Project by Using MeVisLab” is a quick start for module development using MeVisLab.

Section A.2, “Programming Examples” gives an overview of ML programming examples available with the MeVisLab software development kit.

Section A.3, “Exporting Library Symbols” discusses how library symbols are exported to other libraries.

Section A.4, “General Rules for ML Programming” addresses some different issues important for ML module programming, especially to avoid some typical traps and pitfalls when programming ML modules.

Section A.5, “How to Document an ML Module” gives some general documentation hints and tips so that your module fits into the ML module database and into MeVisLab

Section A.6, “Updating from Older ML Versions” describes some compatibility problems and solutions.

Section A.7, “Version Control” explains how different ML versions can be detected.

Appendix B, *Optimizing Image Processing* is a list of hints and approaches to optimize module networks and self implemented ML modules to reach best performance.

Appendix C, *Handling Memory Problems* discusses how to avoid and handle the "Out of Memory" problem.

Appendix D, *Messages and Errors* describes which messages and errors are handled by the ML by the class `ErrorOutput` (Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs”).

Appendix E, *Improving Quality of ML-Based Software* summarizes references to sections which discuss tools, ideas, and classes to improve software quality.

The Glossary is a survey of technical terms used in this document.

2. What You Should Know Before Reading This Document

This document assumes that you are familiar with object-oriented programming in C++. A good knowledge of image processing techniques will facilitate the understanding of the described algorithms. Knowledge of the application MeVisLab, which is strongly related to the ML, is also recommended.

The ML described in this document has the version number 1.8.67.23.86 and is used in MeVisLab 1.6 and later.

Note

Since the ML is still under development, this document is also "work in progress", i.e., some paragraphs may not be completely up to date.

3. Suggestions for Further Reading

For ML and MeVisLab: https://www.mevislab.de

For C++:

- Lippman, S. B., *C++ Primer*, Fourth Edition, Addison-Wesley Longman, 2005.
- Lischner, R., *C++ In A Nutshell, A Language & Library Reference*, O'Reilly, 2003.
- Stroustrup, B., *The C++ Programming Language. Special Edition*, Addison-Wesley Longman, 2000.

For ITK™: http://www.itk.org

For VTK™: http://www.vtk.org

For Open Inventor™:

- Wernecke, J., *The Inventor Mentor*, Release 2, Addison-Wesley, 1994.
- Wernecke, J., *The Inventor Toolmaker*, Release 2, Addison Wesley, 1994.

For Digital Image Processing:

- Gonzalez, R. C., Woods, Richard E., *Digital image Processing*, Second Edition, Prentice Hall, 2002.
- Jähne, B., *Digitale Bildverarbeitung*, 4. Auflage, Springer, 1997.
- Sonka, M., Hlavac, V., Boyle, R., *Image Processing, Analysis, and Machine Vision*, Second Edition, PWS Publishing at Brooks/Cole Publishing Company, 1999.

4. Conventions Used in This Document

There are some textual conventions used in this document:

- This `ClassName` is a class or a module name.
- This `Constant` is a constant or a macro.
- This `Function` is a function or method.
- This *`Parameter`* is a parameter or a function/method argument.

- This `FileName` is a file or a path name.

Additionally, the following pictograms are used:

- Program listings:

```
{
  std::cout << "This is some program code." << std::endl;
}
```

- This is a tip or a useful hint:

Tip

Hey, do it like this! It's better than the other way!

- This is important for understanding and correct programming:

Important

Look at this stuff! It's really important!

- Try to avoid this or do it carefully:

Warning

If you do this, you really should know what you do... it could be dangerous otherwise.

- General notes are shown like this:

Note

This is some interesting additional information.

In many cases, simplified or artificial words will be used without introducing them explicitly if they are self-explanatory.

5. Quick Start

The following chapters are suggested for experienced programmers who want to venture on a quick start in module development without reading this document in detail:

- Start with chapter `Getting Started` of the MeVisLab SDK documentation.
- Continue with Appendix A, *Basics about ML Programming and Projects* to get an overview of important files and module wizards.

Although these chapters contain some redundant information, they provide different approaches to begin module development for MeVisLab and in the ML.

Chapter 1. Conceptual Overview

Chapter Objectives

By reading this chapter, you will get to know

- the basic ML features (see Section 1.1, “Overview”),
- the concepts used in the ML (see Section 1.2, “Principles”),
- a survey of ML classes:
  - the classes used for ML development (see Section 1.3.1, “Classes for Module Development”),
  - the administrative classes (see Section 1.3.2, “Administrative Classes”),
  - the image classes (see Section 1.3.3, “Image Classes”),
  - the helper classes (see Section 1.3.4, “Helper Classes”),
  - the APIs and classes for interaces/voxel type extension (see Section 1.3.5, “APIs and Classes for Interfaces and Voxel Type Extensions”),
  - the component groups (see Section 1.3.6, “Component Groups”),
- and a short overview of modules already implemented or available in the ML Module database (see Section 1.3.7, “The ML Module Database”).

1.1. Overview

The MeVis Image Processing Library (called ML in the following) represents a general approach to image processing. It is based on the following principal ideas:

- Image processing algorithms are represented by modules (sometimes also called operators or nodes).
- Modules are mainly arranged in a directed graph that represents the flow of image data.
- Modules implement a unified image processing interface.
- Modules are self-descriptive by exporting their parameters as fields (field interface).
- Image data is processed in fractions, i.e., page by page (paging).
- Pages can be processed in parallel if supported by a module (multithreading).
- Image processing is performed on a request-oriented basis (pull model, processing on demand).
- Images can have up to six dimensions.
- Image pixels (called voxels in this document) can be single scalars or structures (e.g vectors, complex values or matrices).
- Platform independence, pure C++ code running on Windows, Linux and Mac OS X.
- A C interface of the ML is available for applications that do not use C++.

1.2. Principles

When you start using the ML, you should understand the most important ML properties and features.

1. **Modules and `Host`**

   Generally speaking, the ML is a C++ library that contains many classes for efficient image processing.

   This requires a number of image processing algorithms derived from the module base class `Module` (called *Modules*), and an entity that controls these modules and organizes the data transfer between them - the `Host` . Each module has a self-descriptive parameter interface built of so-called *Fields* that also automatically implements module persistence.

2. **Module Networks**

   For complex image processing tasks, many image processing steps need to be combined:

   - To solve an image processing problem all required modules need to be combined to a *module network* that defines the image data flow as a directed module graph.
   - During image processing, the image data flow is realized as a flow of image fractions, the so-called *pages* which can be regarded as rectangular subimages that contain image data.
   - The image data flow is *demand-driven*, i.e., the module network processes only pages that are needed for output image computing. This concept is often also called *pull-driven* or *request-driven*.

3. **Image Model**

   An ML image has the following properties:

- It has up to *six dimensions* (x, y, z extent, color (c) dimension, time (t) dimension and user (u) dimension).
- Although color and *color channels* are supported (4th dimension), the ML is generally *color format independent*.
- Image pixels (called *Voxels* for **Vo**lume Pi**xel**) can be *simple scalars* or *structured* elements.
- Generally, an image consists of a set of *properties* (extent, data type, page extent, medical information, voxel extent, DICOM tags, etc.) and
- a set of image data fractions (pages) in memory.

4. **Advanced Features**

   The ML offers some special features to support advanced image processing:

   - Intermediate results are automatically stored in a buffer to avoid recalculations (*caching*).
   - Pages can be processed in parallel during image flow processing (if supported by the modules). See Section 3.1.11.3, “Multithreading: Processing Image Data in Parallel” for more information.
   - Classes for *Error Handling*, *Debugging* and *Memory Management* are available.
   - The ML is *platform independent*, i.e. it runs on Microsoft®, MAC and Linux platforms, and should be portable to other platforms without much effort.
   - All C++ code of the ML is written in its own *namespace* to protect it from collisions with other libraries.
   - A **C**-**A**pplication **P**rogramming **I**nterface (*C-API*) is available to provide the ML to other programming languages.
   - A *Runtime Type System* provides a factory for all important classes, modules and types.

5. **Class Groups**

   The ML provides a set of classes that can be divided into three groups:

   - *Core classes* that build the basics for image processing and module handling
   - *Helper* or *user classes* that simplify the implementation of algorithms
   - *Aggregated projects* that are not necessarily part of the ML, but essential or very useful for advanced image processing.

6. **Module Database**

   There is a large number of modules that cover many image processing tasks:

   - Input/Output, Arithmetic, Segmentation, Morphology, Geometry, Statistics, Medical Analysis, Drawing, Diagnosis, and many other.
   - When ML is used in conjunction with MeVisLab, a large number of
     - modules for ML *image visualization* (2D and 3D)
     - modules for standard *3D scene modeling* and
     - *macro modules* for complex image processing and visualization tasks

     is also available.

1.3. ML Classes - Overview

The ML does not only implement base classes for image processing, but also covers areas such as linear algebra basics, error handling, debug functionality, a runtime type system, runtime types for voxels, observable parameter classes, containers for fields and much more.

The following sections will give a short overview of most of these classes and will explain their purpose.

1.3.1. Classes for Module Development

This chapter gives a survey of the most important ML components, classes and interfaces.

1.3.1.1. `Module` Overview

The class `ml::Module` is the base class from which all image processing modules need to be derived to implement new algorithms for image processing. It provides a number of virtual methods which can be overloaded to implement and control image processing, and to handle/add/change algorithm and processing parameters. See Chapter 3, *Deriving Your Own Module from Module* for more information.

1.3.1.2. `Field` Overview

The `ml::Field` class encapsulates a data type such as an integer, a vector, a matrix, a string or even an image or a complex data structure. Currently, about 60 field types are available in the ML. A field is useful for various purposes: It can be observed for changes (listener pattern), its state can be saved/restored by handling its value as a string, and it can be connected with fields of other modules for data transfers. Modules use these features for creating a reflective, self-descriptive and persistent parameter interface. See Section 2.1.2, “ `Field` ” for more information on fields.

1.3.1.3. `FieldContainer` Overview

The `ml::FieldContainer` class manages a list of fields. A module is derived from the field container, so all modules manage their fields themselves. See Section 2.1.3, “ `FieldContainer` ” for more information.

1.3.1.4. Image Classes Overview

The ML does not process images as a whole but breaks them down into smaller fractions of identical extents, the so-called *pages* (see Section 2.3.4, “ `PagedImage` ”). Pages can be easily buffered, cached and processed in parallel without spending too much memory or time. The ML cache stores pages that have a chance to be reused. For algorithms that cannot easily be implemented on a page basis, the ML provides specialized classes that also use internal paging, if possible. See Section 2.3.7, “ `VirtualVolume` ”, Section 2.3.6, “ `BitImage` ”), Section 2.3.8, “ `MemoryImage` ” and Section 1.3.3, “Image Classes” for details.

1.3.2. Administrative Classes

1.3.2.1. The `Host` Overview

The `ml::Host` is the core class of the ML. It manages the entire image processing workflow including on-demand, page-based image processing, caching and parallelization as well as calling module functionalities for image processing. It also provides functions such as checking and updating module graphs, calculating (sub-) images with `getTile()` commands and caching intermediate results by using the `MLMemoryManager` (see Section 1.3.2.2, “The `MLMemoryManager` and Memory Handling”). It checks and processes a set of ML modules that have been derived from the class `Module` (see Chapter 3, *Deriving Your Own Module from Module*) and are connected forming a directed acyclic graph.

1.3.2.2. The `MLMemoryManager` and Memory Handling

The `MLMemoryManager` is dedicated to managing a certain buffer memory (the *Page Cache*) where pages that have been generated by an image processing module are stored for later reuse. If a given cache limit is exceeded, the memory manager frees memory blocks using a least-recently-used caching strategy until the cache limit is no longer exceeded. The MLMemoryManager provides strong and weak smart pointers to keep reference to managed memory blocks and is not limited to the use of ML pages, can be used by other application libraries as well and allows to make libraries cache-aware. Memory blocks that are locked by a smart pointer will not be removed, so it is possible that the cache size exceeds the limit temporarily.

1.3.2.3. `Memory` Overview

Since the ML is dedicated to efficiently processing images that are too large for being stored in memory, the ML controls memory handling globally in order to allow for a safe and efficient memory usage.

The ML class `Memory` currently provides functions for memory allocation, reallocation, freeing, etc. It contains only basic functionality. In future versions, it will use automatic strategies to (re-) organize and/or clean up the memory (and the ML cache) to reduce or prevent out-of-memory errors. Allocation errors can be handled by the ML or by users in different ways. See Section 1.3.2.3, “`Memory` Overview” for more information.

1.3.2.4. The Runtime Type System

The Runtime Type provides an interface where all important classes and modules can register themselves (see Section 2.2.4, “The Runtime Type System” for more information). It stores class names, types, source library, parent classes, etc. It also provides a factory that permits the creation of instances of any registered (non-abstract) class from a class name string. It is crucial for applications such as MeVisLab that need to handle databases and networks of modules not yet known when the application is compiled. These concepts are realized by the classes `Runtime`, `RuntimeDict` and `RuntimeType` as well as by a set of macros that implement runtime module interface functionality.

1.3.2.5. Debugging Overview and Error Handling Support

The ML supports debugging and error handling (see Chapter 5, *Debugging and Error Handling* for more information).

Debug output can be controlled in the ML, i.e., it can be enabled/disabled for the entire ML or activated for individual classes.

Errors can be handled on different levels. In general, programmers should check for errors and handle them by using dedicated error handling macros and *never* by using statements such as `assert`, `abort` or `exit`, because the application cannot manage or log these statements. The way how the ML behaves on errors can be configured globally. The ML could generate an e-mail and terminate, or display a pop-up window and try to continue. The behavior on errors should always be configured globally for the ML.

The classes `ErrorOutput` and `ErrorOutputInfos` are used for error handling and ML output redirecting. They contain a set of static functions to print debug information, warnings, errors and fatal errors. There is a registration mechanism where the application can register itself to be notified when an error, a warning or some debug information is to be printed or handled.

1.3.3. Image Classes

1.3.3.1. `ImageProperties` Overview

The `ml::ImageProperties` class describes the basic image properties 6D image extent, voxel data type, and minimum and maximum voxel values. See Section 2.3.1, “ `ImageProperties` ” or `mlImageProperties.h` in project `ML` for more information.

1.3.3.2. MedicalImageProperties Overview

The ml::MedicalImageProperties class is derived from ImageProperties. It contains additional information specialized for medical data sets, like voxel size, image orientation and position, a reference to the image's DICOM information and descriptions of color channels, time points and the user dimension. See Section 2.3.2, "MedicalImageProperties" or mlMedicalImageProperties.h in project ML for more information.

1.3.3.3. PagedImage Overview

Since images are usually **not** processed as a whole by the ML, it is necessary to break them down into smaller fractions of identical extents, the so-called *pages*. The ml::PagedImage class is derived from the class MedicalImageProperties (see Section 2.3.2, "MedicalImageProperties" and thus inherits all properties that describe an image. It is dedicated to managing paged images in the ML and also to representing image outputs of ML modules. See Section 2.3.4, "PagedImage" or mlPagedImage.h in project ML for more information.

1.3.3.4. SubImage and TSubImage Overview

This class represents image, subimage and page buffers, hence knowledge about this class is crucial for programming most image processing algorithms. It manages chunks of voxel data, copies or fills them and offers fast data access methods with offset/stride usage. ml::TSubImage is the typed version of ml::SubImage which permits typed data access. See Section 2.3.5, "SubImage/TSubImage", or mlSubImage.h in project ML.

1.3.3.5. VirtualVolume

For many algorithms, the implementation of a page-based approach might be difficult. A typical example is a filling algorithm that needs random access to the input image, even if only one voxel of the output image is required. Such algorithms can be implemented efficiently by using the ml::VirtualVolume and ml::TVirtualVolume classes (see Section 2.3.7, "VirtualVolume") without breaking the page-based approach of the ML. A VirtualVolume class requests/calculates the image data when needed and rejects it if not needed anymore. Thus large images can randomly be accessed without the need to keep them completely in memory.

1.3.3.6. BitImage

The ml::BitImage class (see Section 2.3.6, "BitImage") supports the memory-efficient creating, copying, filling, addressing of packed 6D binary images, and interactions with the class SubImage and VirtualVolume (see Section 2.3.5, "SubImage/TSubImage" and Section 2.3.7, "VirtualVolume".) This is often useful for algorithms that need to mark or tag all voxels of a page or an image, or for those algorithms that simply need to handle large binary images.

1.3.3.7. MemoryImage

Algorithms that need access to an entire non-paged memory-mapped image may use the MemoryImage approach (see Section 2.3.8, "MemoryImage") for image processing. This approach breaks the paging principle and should only be used if it cannot be avoided and if it is safe to load the whole image into memory. The memory image is integrated as a special member of the class PagedImage (see Section 2.3.4, "PagedImage") and can thus be used in parallel or instead of a paged image.

1.3.4. Helper Classes

1.3.4.1. ImageVector

The class ml::ImageVector manages a 6D point or vector with integer components and is used for voxel positions, image extents, page extents and boxes. The typical (integer) vector

arithmetic is available as well as are methods for minimum and maximum component determination, lexicographical comparisons, stride operations and component multiplication for voxel addressing, etc. See Section 2.4.1, "`ImageVector`, ImageVector" for more information.

1.3.4.2. `SubImageBox`

The class `ml::SubImageBox` manages a rectangular 6D box specified by two integer `ImageVectors` that represent its corners. It permits intersections, calculation of voxel volumes, etc. (see Section 2.4.2, "`SubImageBox`"). Like the class `ImageVector` (see Section 2.4.1, "`ImageVector`, ImageVector"), it is available in 16, 32 and 64 bit template specializations. The default version `SubImageBox` also uses 64 bit integer addressing. An analogous class that uses 6D double (Vector6) vectors is available as `SubImageBoxd` in the file `mlSubImageBoxd.h` .

1.3.5. APIs and Classes for Interfaces and Voxel Type Extensions

An easy way to use the ML is to link the C-API (see Section 6.3, "mlAPI.h") of the ML. It provides a set of functions to create/delete modules, set their parameters (fields), to connect them and to request images from their outputs. The C-API allows non-C++ applications to make use of the ML. This interface is more stable than the C++ interface which may be modified more frequently and is not guaranteed to be binany compatible between ML versions.

The ML supports a set of scalar data types for image voxels (signed/unsigned 8, 16, 32, 64 bit integer types and float, double) as well as extended data types (see Chapter 7, *Registered Voxel Data Types*, Section 7.5.5, "Implementing a New Voxel Data Type by Deriving from MLTypeInfos") that permit the usage of composite data types added by the user or by the ML itself. It is not necessary to recompile the ML to make use of these types, but modules might need to be adapted depending on how they were written.

1.3.6. Component Groups

ML classes can roughly be divided into three groups - *Core Classes*, *Helper* or *User Classes* and *Aggregated Projects and Classes*. Note that these classes are not necessarily in the same directory or project. Also note that some of these classes are not directly located in the project ML. Basic functionality is typically located in the project `MLUtilities` (see Section 2.6.2, "`MLUtilities`", vector and matrix arithmetics are located in the project `MLLinearAlgebra` (see Section 2.6.1, "`MLLinearAlgebra`(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)" and classes related to image processing are located in the project `ML`.

- *Core classes* that provide the basic functionality for image processing and module handling:
  - `Module` the base class for image processing modules,
  - `Field`, `FieldContainer` and `FieldSensor` for module parameters,
  - `Host`, `MLMemoryManager` for image data and image flow management,
  - `ImageProperties` and `MedicalImageProperties` for image properties,
  - `ImagePropertyExtension` that can be appended to medical image properties,
  - `PagedImage` and `MemoryImage` for image handling,
  - `SubImage` / `TSubImage` for image, subimage and page handling,
  - `ErrorOutput` (from project Section 2.6.2, "`MLUtilities`") for error handling and logging,
  - `InputConnector`, `OutputConnector` for module input/output image connections.

- *Helper* or *user classes* and projects simplifying the implementation of some algorithms:
  - `Runtime`, `RuntimeDict` and `RuntimeType` (from project Section 2.6.2, “ `MLUtilities` ”) for object factories and class management,
  - `ImageVector` and `SubImageBox` for position and region management,
  - Linear algebra classes like `Vector2`, `Vector3`, `Vector4`, `Vector6`, `Matrix3`, `Matrix4` (from project `MLLinearAlgebra`),
  - `MLTypeInfos` for registered and user-defined data types (see Chapter 7, *Registered Voxel Data Types*),
  - `Rotation`, `Disc`, `SubImageBoxd`, `Line`, `Plane` parameters for modules,
  - `DateTime`, `TimeCounter`, `Notify` (from project Section 2.6.2, “ `MLUtilities` ”) for high precision time measurement, and
  - `ScaleShiftData` for image data scaling.
- *Aggregated projects and classes* that are not necessarily part of the ML but that are essential for advanced image processing:
  - `BitImage` for flag and mask images,
  - `MLBase` for point, function, marker, and vector lists, etc. that can be passed between modules,
  - `MLDataCompressor` for data compressor classes,
  - `MLDiagnosis` with support for module tests and inspection,
  - `MLImageFormat` for ML specific file IO,
  - `VirtualVolume` for random access to (large) paged images,
  - and many more.

1.3.7. The ML Module Database

Many modules that use the ML are already available to developers:

- Image file IO and DICOM support,
- Arithmetic (Add, Subtraction, Inversion, And, Or, Xor, Sqrt, Sqr, Log, Exp, etc.),
- `Base` objects for marker, vector, point or general object lists,
- FlowControl (ImageIteratorStart/End, Switch, Bypass, etc.),
- Geometry (image resizing, subimages, resampling, concatenation, MPR, etc.),
- Distance and projection transformations (Radon, DistanceTransform, etc.),
- Segmentation (medical imaging: LiveWire, vessels, tumors, region growing, thresholding, fuzzy),
- Registration and image matching,
- Drawing (RasterFunctions, Draw2D, Draw3D, LiveWire, etc.),
- Statistical (GlobalStat, CalcVolume),
- Morphology (Rank, Min, Max, Median, Gauss, Average, Statistical, Laplace, Edge detectors, CloseGap, Surround, etc.),

- Transfer functions (Look up tables),
- Object (list) handlers,
- Color (model) management (color converters and tables),
- Diagnostic modules for controlling ML core functions and for error and debug handling,
- Texture analysis filters,
- Logfile, tester and inspector modules,
- and many more.

When using the ML in combination with MeVisLab, a large number of non-image processing modules can also be used for module networks and applications:

- Viewers (2D, 3D, Shadow, Slab, Slice Viewer, Volume Rendering, etc.),
- Vessel visualization,
- 2D/3D object list and marker managers and visualizers,
- Transfer functions (Look up tables, LUT),
- Diagram visualization (draws 2D coordinates, markers, points, lines, etc.),
- **C**ontour **S**egmentation **O**bjects (CSO) for contour drawing, manipulation and conversion,
- **W**inged **E**dge **M**esh library (WEM) for iso surface management and surface shaded display,
- Interactions (View2DExtensions, Manipulators, Draggers, MarkerEditor, etc.), and
- a set of (helper) macros (e.g., convenience viewers, frequently used module groups, converters between scalars, vectors, matrices, etc.).

The following scientific packages (each of them offering hundreds of algorithms) are available:

- `Open Inventor ™`: A set of modules (nodes) for 3D rendering with cameras, transformations, 3D viewers, textures, 2D and 3D text, manipulators, shape objects, etc.
- `The Insight Segmentation and Registration Toolkit ™`: Algorithms for advanced image processing, registration, and image analysis.
- `The Visualization Toolkit ™`: Algorithms for advanced visualization, rendering, and image processing.

Chapter 2. Detailed Class Overview and Usage

Chapter Objectives

This chapter will give you a detailed overview of most ML classes and tools:

- Classes for Module Development
  - `Module`, the base class from which new modules are derived,
  - `Field`, a class to encapsulate, manage and observe module parameters,
  - `FieldContainer`, a parent class of module to manage a set of fields.
- Classes for administering ML internals as well as for managing `Base` objects and modules
  - `Host`, the core class managing image processing, data flow, parallelization, etc.,
  - `MLMemoryManager`, library to store temporary (image) pages for reuse,
  - `Memory` providing an interface for reliable memory allocation and error handling on failures,
  - `Base`, the base class for most ML classes, e.g., to implement a general persistence concept,
  - The Runtime Type System classes `Runtime`, `RuntimeType` and `RuntimeDict` for object factories and type management,
  - Debugging Classes.
- Helper Classes for 6D (voxel) positions and subimage boxes (Section 2.4.1, “`ImageVector`, ImageVector”, Section 2.4.2, “`SubImageBox`”
- Image Classes to describe images, image properties (Section 2.3.1, “`ImageProperties`”, Section 2.3.2, “`MedicalImageProperties`”, Section 2.3.3, “`ImagePropertyExtension`”), subimages (Section 2.3.5, “`SubImage`/`TSubImage`”) and paged images (Section 2.3.4, “`PagedImage`”
  - `BitImage` to handle packed flag and binary images,
  - `VirtualVolume` for global page-based image processing,
  - `MemoryImage` for special cases that require completely memory-mapped images.
- Tools and helper classes to support advanced image processing
  - Some `Vector` and matrix arithmetics (`MLLinearAlgebra`),
  - Classes and tools for kernel-based image processing (Section 2.6.5, “`MLKernel`”),
  - A tool box with a set of frequently used helper functions (Section 2.6.6, “`MLTools`”).
  - Several diagnosis modules (Section 2.6.7, “`MLDiagnosis`”),
  - Classes to save, modify, load and query ML image data in a dedicated ML file format (Section 2.6.8, “`MLImageFormat`”),
  - Data compressor classes (Section 2.6.9, “`MLDataCompressors`”).
- Classes that contain additional voxel data types (RegisteredDataTypeClasses)

- Important data types used in ML sources (Section 2.8, “ML Data Types”).

2.1. Classes for Module Development

2.1.1. `Module`

`ml::Module` is the base class all ML modules are derived from. It is crucial to know this class for being able to extend the ML module database. See Chapter 3, *Deriving Your Own Module from Module* for a detailed description of this class and how to derive own modules from it.

2.1.2. `Field`

A field is a C++ class which simply encapsulates a data value such as an integer, a vector, a matrix, a string or even an image or a complex data structure. All fields are derived from the `ml::Field` base class. Currently, about 60 field types are available in the ML. Due to the following reasons, fields may be considered as one of the most powerful ML features:

- A field's value can be set/retrieved as a string value from the base class, but each derived version has also set/get methods for typed values. The *string value methods* permit an application to work with field values without considering the field type.
- Fields can be *observed*. Each time a field is modified, all observers are notified and can adapt themselves to the new field value. A standard observer is, for example, the module that contains the field. Other observers could be the user interface (of the module using the field) or other fields that update themselves if the value changes.
- A field can be attached to other fields. Whenever the field is changed, the connected fields are notified and/or the new value is transferred to the connected field. In module networks, this is a powerful feature to hard-code information flows, i.e., to define how modules can communicate with each other. This heavily reduces code complexity of applications that use such networks.
- Since fields support setting/reading their values as strings, all fields can accept values from other fields. A transferred value is scanned as far as possible, i.e., an integer value will accept float values (only the integer fraction) and vice versa. But also vectors accept integers, doubles, floats and vice versa; enumerated values can be set as integers, etc.
- All ML modules use fields for their parameter interface. Each module includes a field container which can handle an individual list of fields. There is no need to know set or get methods to communicate with a module. Hence, the application MeVisLab can simply ask the field container of a module to return all fields, their names and their values, and can automatically create a user interface for the module or it can save/load the field values for persistence.

2.1.2.1. Standard Fields

The following derived field classes are part of the ML:

- `BaseField` - Contains a pointer to any `Base` object (Section 2.2.3, “ `Base` ”). Using `BaseFields`, arbitrary `Base` data structures can be shared between modules.
- `BoolField` - Contains a Boolean value.
- `ColorField` - Contains an RGB color value.
- `FloatField`- Contains a floating point number.
- `DoubleField` - Contains a floating point value with double precision.
- `EnumField` - Contains a list of strings that represents an enumeration value as well as an index to the currently selected entry.

- `InputConnectorField` - Contains an input connector value which can be connected to the output connector value of other modules in order to establish image connections between modules.
- `IntField` - Contains a 64 bit integer value of type `MLint`.
- `MatrixField` - Encapsulates a `Matrix4` value (Section 2.6.1, "MLLinearAlgebra(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)"), a 4x4 matrix of `double-type` numbers.
- `NotifyField` - Does not represent a value; it is used to propagate field changes or to implement buttons on user interfaces.
- `OutputConnectorField` - Encapsulates an output connector of a module used for connections with input connectors of other modules in order to establish image data flows between modules. Note that this field also contains a `PagedImage` (Section 2.3.4, " PagedImage ") to manage the paged output image of a module.
- `PlaneField` - Encapsulates a plane described by a plane equation in 3D space.
- `RotationField` - Encapsulates a rotation value described as quaternion of four floating point values.
- `SoNodeField` - Encapsulates a pointer to an Open Inventor™ node to make scene graphs available at module outputs, e.g., for visualization purposes.
- `StringField` - Encapsulates a standard string value.
- `SubImageBoxField` - Encapsulates a `SubImageBox` value (Section 2.4.2, " SubImageBox ") with corners specified by integer vectors (Section 2.4.1, "ImageVector, ImageVector").
- `SubImgBoxdField` - Encapsulates a `SubImageBoxd` (Section 2.6.3.1, " SubImageBoxd ") with corners specified by floating point vectors `Vector6` (Section 2.6.1, "MLLinearAlgebra(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)").
- `Vector2Field` - Encapsulates `Vector2` value (Section 2.6.1, "MLLinearAlgebra(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)") .
- `Vector3Field` - Encapsulates `Vector3` value (Section 2.6.1, "MLLinearAlgebra(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)").
- `Vector4Field` - Encapsulates a `Vector4 value` (Section 2.6.1, "MLLinearAlgebra(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)").
- `Vector6Field` - Encapsulates a `Vector6 value` (Section 2.6.1, "MLLinearAlgebra(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)").
- `ImageVectorField` - Encapsulates a `ImageVector` value (Section 2.4.1, "ImageVector, ImageVector").
- `MLDataTypeField` - Encapsulates `MLDataType(` value Section 2.6.3.2.1, "MLDataType").
- `ProgressField` - Contains a floating point value in `[0...1]` which can be incremented or reset to represent a progress/done indicator.
- `UniversalTypeField` - Contains an arbitrary value of any of the available ML data types (Section 2.6.3.2.1, "MLDataType"). The managed type can be changed on runtime.

See the file `mlFields.h` for a detailed description of the `Field` classes.

2.1.2.2. Important `Field` Methods

The following list gives an overview of the most important `Field` methods:

- `void setStringValue(const std::string &value)`

  Assigns the string *`value`* to the field.

- `std::string getStringValue() const`

  Returns the value of the field as standard string.

- `void touch()`

  Simulates a change of the field value so that all attached fields or field sensors are notified. Note that this method has no effect if notification handling (for this field or globally) is disabled.

See the file `mlFields.h` for further methods of the `Field` class.

2.1.2.3. `Base` Field

The `ml::BaseField` is used to transfer arbitrary data from one module to another and to check it there for its correct type. Do the following:

- Derive a data object from the class `Base`.
- Add a `BaseField` to your module (Section 2.1.3, “ `FieldContainer` ”).
- Use the method `setBaseValue()` to set the address of your `Base` object as a value of the `BaseField`.
- Add a `BaseField` to another (second) module and connect both fields.
- In the second module, use the `getBaseValue()` to retrieve the `BaseField` value.
- Cast the value of the base field with `mlbase_cast<BaseType*>(fieldValue)` to the target type `BaseType`. If the `fieldValue` is of a different type than `BaseType*`, `NULL` is returned by the cast.

Do not forget

- to initialize each new `BaseType` subclass `MyNewBaseField` with a `MyNewBaseType::initClass()` statement in the library initialization and
- to use "input" or "output" as first part of the field name to define the field as an input or output of your module.

The following examples show how to

- derive a new class from `Base`,
- create a module and to export an internal `Base` object via a `BaseField`,
- create a module to which any `Base` object can be connected, and
- check whether a `Base` object is of a certain or correct type.

Example 2.1. Deriving a New Class from `Base`

```
ML_START_NAMESPACE

class ML_BASE_EXAMPLE_EXPORT MyBaseObject : public Base
{

public:

  // Constructor which initializes the internal string.
  MyBaseObject() { _strValue = ""; }

  // Return value of string.
  const std::string &getValue() const { return _strValue; }

  // Set string value.
  void setValue(const std::string &strVal) { _strValue = strVal; }

  /*...*/

private:

  // Contents of your base object, in this case a string.
  std::string _strValue;

  // Define the interface of the class to the runtime type system of the ML.
  ML_CLASS_HEADER(MyBaseObject);
};

ML_END_NAMESPACE

// ***Source file: Implement the class interface for the runtime type system.
ML_START_NAMESPACE

  ML_CLASS_SOURCE(MyBaseObject, Base)

ML_END_NAMESPACE
```

Example 2.2. Making a `Base` Object Accessible via a `BaseField`

```
// ***Header File in class definition:

ML_START_NAMESPACE

class Export MyModule : public Module
{

public:

  MyModule::MyModule();

  // Create a base object which shall be passed to another module via a base field.
  MyBaseObject objectToTransfer;

  //! Handle field changes of the field field.
  virtual void handleNotification (Field *field);

private:

  // Pointer to a base field to which any base object can be connected.
  BaseField* _baseFld;

  // Define the interface of the class to the runtime type system of the ML.
  ML_MODULE_CLASS_HEADER(MyModule);
};

ML_END_NAMESPACE
```

Example 2.3. Getting a `Base` Object from a `BaseField` Connection and Checking its Type

```
// ***Source file:Construct the module.

ML_START_NAMESPACE

ML_MODULE_CLASS_SOURCE(MyModule, Module)

// Construct the module and initialize the objectToTransfer with a test string.
MyModule::MyModule() : Module(0,0)
{
  handleNotificationOff();

  /*...*/

  // Initialize base object
  objectToTransfer.setValue("TestString");

  // Add a base field and set the address of objectToTransfer as its value.
  _baseFld = addBase("outputBase")
  _baseField->setBaseValue(&objectToTransfer);

  /*...*/

  handleNotificationOn();
}


void MyModule::handleNotification(Field * fields)
{
  if (_baseFld == field){

    // Base field has changed, get its value.
    Base *baseValue = baseFld->getBaseValue();

    Cast to target type and by that, check for correct type.
    MyBaseObject* myBO = mlbase_cast<MyBaseObject*>(baseValue);

    // Check for validity.
    if (myBO != NULL){

      // Print value
      mlDebug(myBO->getValue().c_str());
    }
  }
}

ML_END_NAMESPACE
```

2.1.3. `FieldContainer`

The ml::Module is derived from ml::FieldContainer which encapsulates a list of fields for the module (see also class `Field`). So the `Module` provides field list access, removal, search and indexing. In a `Module` constructor, all interface parameters of the modules are added to this container. The most important methods are:

- `Field* addField(const char* name,const char* type, const char* value, bool createSensor = true)`

  Adds a new field with name *`name`*, type *`type`* and value *`value`* (coded as a string) to the container. If *`createSensor`* is `true` (the default), the `Module` of the field container will be installed as an observer of the field so that field changes are passed to the `Module` as notification.

- `Field* addField(Field* field, bool createSensor = true)`

  Adds the field *`field`* and installs the `Module` as an observer if *`createSensor`* is `true` (the default) so that field changes are passed to the `Module` as a notification.

- `Field *add[Bool | Int | Enum | Float | Progress | Double | String | Notify | InputConnector | OutputConnector | Base | SoNode | DicomTagList | Vector2 | Vector3 | Vector4 | Vector6 | ImageVector | SubImageBox | SubImageBoxd | Plane | Rotation | Color | Matrix | MLDataType | UniversalType] (std::string name, ...)`

Creates a field of the specified type with the name *name*, adds it to `Module` and installs the `Module` as an observer so that its `handleNotification()` method is called on field changes.

- `Field* getField(std::string name) , Field *getField(int index)`

  Returns the `Field` with name *name* or at index *index*; returns `NULL` if not found.

- `int getField(Field *field)`

  Searches field with address *field* in list and returns its index. If not found, `0` is returned and a warning is sent to the ML error handler.

- `int getSize()`

  Returns the number of fields in the container.

- `int getNumInputConnectorFields() const`

  Returns the number of added `InputConnectorFields`.

- `int getNumOutputConnectorFields() const`

  Returns the number of added `OutputConnectorFields`.

- `InputConnectorField *getInputConnectorField(int i) const`

  Returns the i^{th} `InputConnectorField` in the container. If not found, `NULL` is returned and `ML_FATAL_ERROR` is sent to the ML error handler.

- `OutputConnectorField *getOutputConnectorField(int i) const`

  Returns the i^{th} `OutputConnectorField` in the container. If not found, `NULL` is returned and `ML_FATAL_ERROR` is sent to the ML error handler.

- `std::string getValue(const std::string &name) const`

  Returns the value of a field with name *name* as standard string. If the field is not found, a warning is sent to the ML error handler and an empty string ("") is returned.

- `std::string getValue(int index) const`

  Returns the value of a field at position *index* in the container as standard string. If the field is not found, a warning is sent to the ML error handler and an empty string ("") is returned.

- `setValue(int i, const std::string &value)`

  Assigns value *value* to a field at position *index* in container. If the field is not found, a warning is sent to the ML error handler and no value is assigned.

- `setValue(const std::string &name, const std::string &value)`

  Assigns value *value* to a field with name *name* in container. If the field is not found, a warning is sent to the ML error handler and no value is assigned.

- `setValue(const std::string &name, int value)`

  Assigns the integer *value* to a field with name *name* in container. If the field is not found, a warning is sent to the ML error handler and no value is assigned.

- `void activateAttachments()`

(Re)Enables the notification of attached fields and field sensors when field values are set or notified by e.g. `touch()` or `set*Value()` methods.

- `void deactivateAttachments()`

  Disables the notification of attached fields and field sensors when field values are set or notified by e.g. `touch()` or `set*Value()` methods.

2.1.4. Image Classes for Module Development

See Section 2.3, “Image Classes” for a detailed image and subimage handling description.

2.2. Administrative Classes

2.2.1. `Host`

The class `Host` manages the entire image processing in the ML including paging, caching, parallelization and calling any `Module::calculate*` functionality. It also provides functions such as checking and updating module graphs as well as calculating (sub)images with `getTile()` commands.

The `Host` processes a set of ML modules that are derived from the class `Module` and connected as a directed acyclic graph.

Note

Do not try to use the `Host` directly by using its methods or functions. All of its important functionality is wrapped as static functions in `Module`. The `Host` should remain a "hidden" part of the ML so that `Host` replacements and improvements do not endanger module compatibility.

For memory-optimized calculation of an image or subimage, each ML module (derived from `Module`) supports so-called *page-based* image processing, i.e., each image is divided into blocks (called *pages*) which are then processed sequentially or in parallel and are finally combined to the requested (sub)image.

Consequently, the memory usually does not hold the complete image, but only the currently used fragments.

The page extent is defined as an image property of each ML module output. Page-based image processing can degenerate to global image processing when the page extent is set to the extent of the actual image. This, however, is the worst case and should be avoided.

During image processing, the ML stores as many pages as possible in the cache of the `MLMemoryManager` (Section 1.3.2.2, “The `MLMemoryManager` and Memory Handling”) to reach maximum performance. Repeated (sub)image requests can often be processed more efficiently by simply reusing existing pages. The extent of pages can be controlled by the application or by the user.

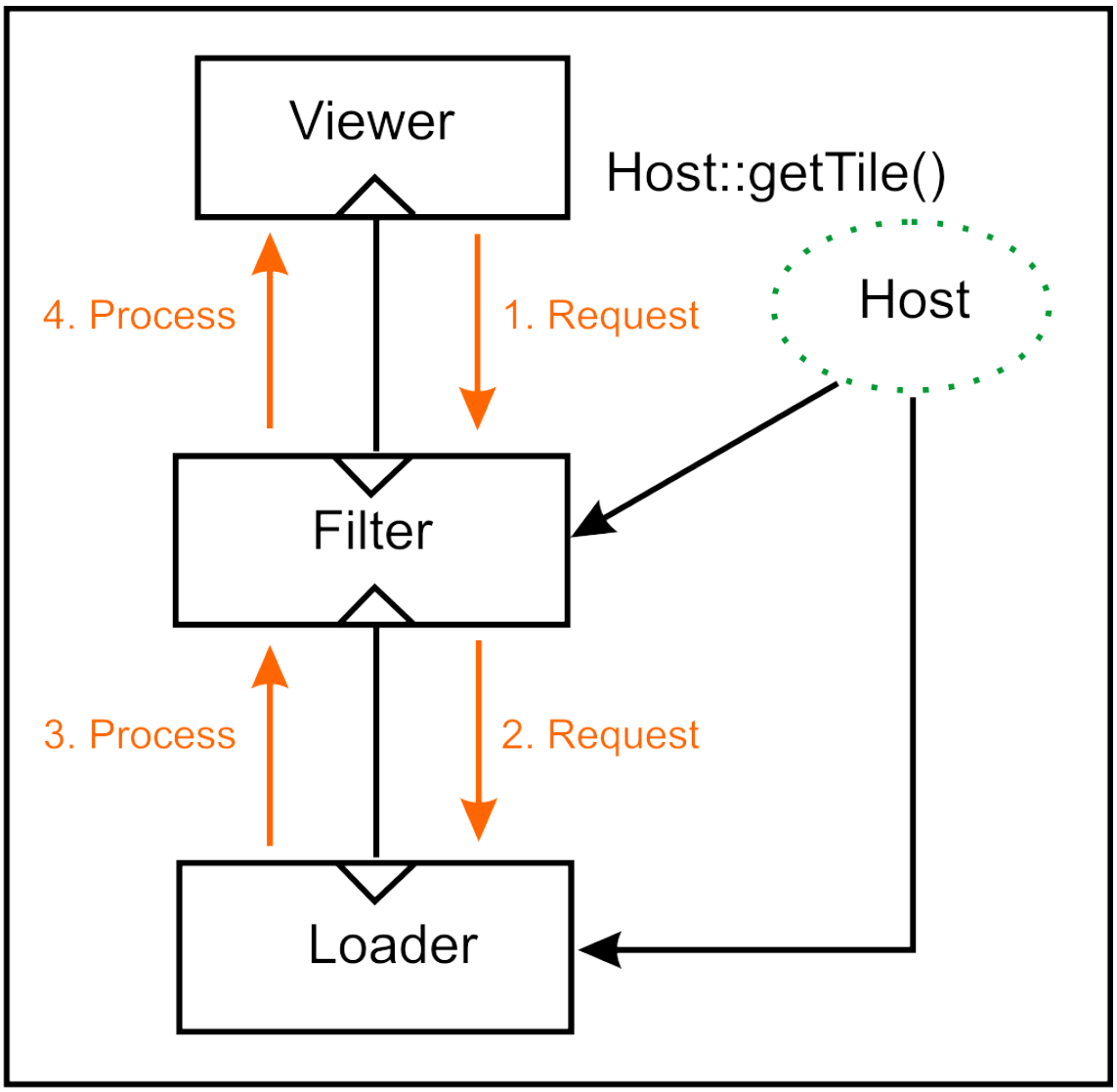
Overview of image requests performed by the `Host`:

1. `Viewer` shows image properties and data from `Filter`.

2. `Filter` calculates results from image properties and data from `Load`.

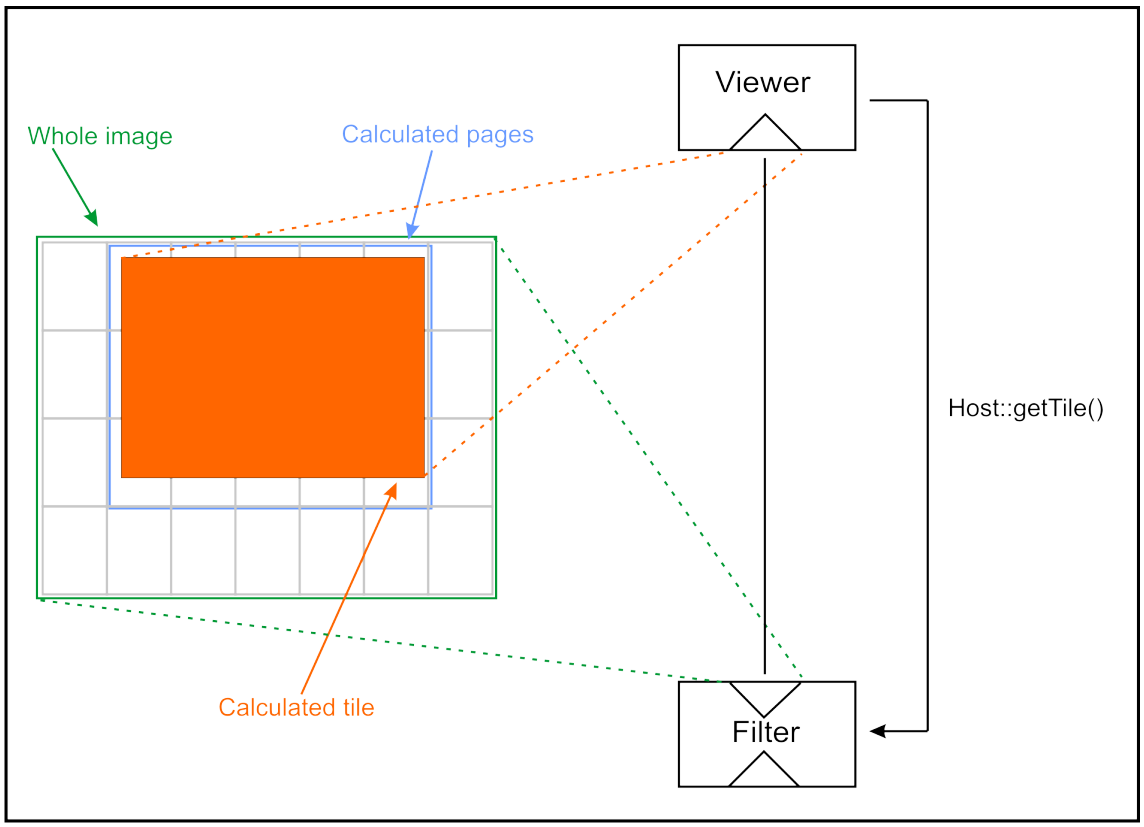
The `Host` remains invisible to the module but processes all `Viewer` requests. Functions (e.g., getTile()) are wrapped in the class `Module`:

Figure 2.1. Requesting an Image Pipeline with getTile()



When a tile is requested, the tile request is broken down into page request:

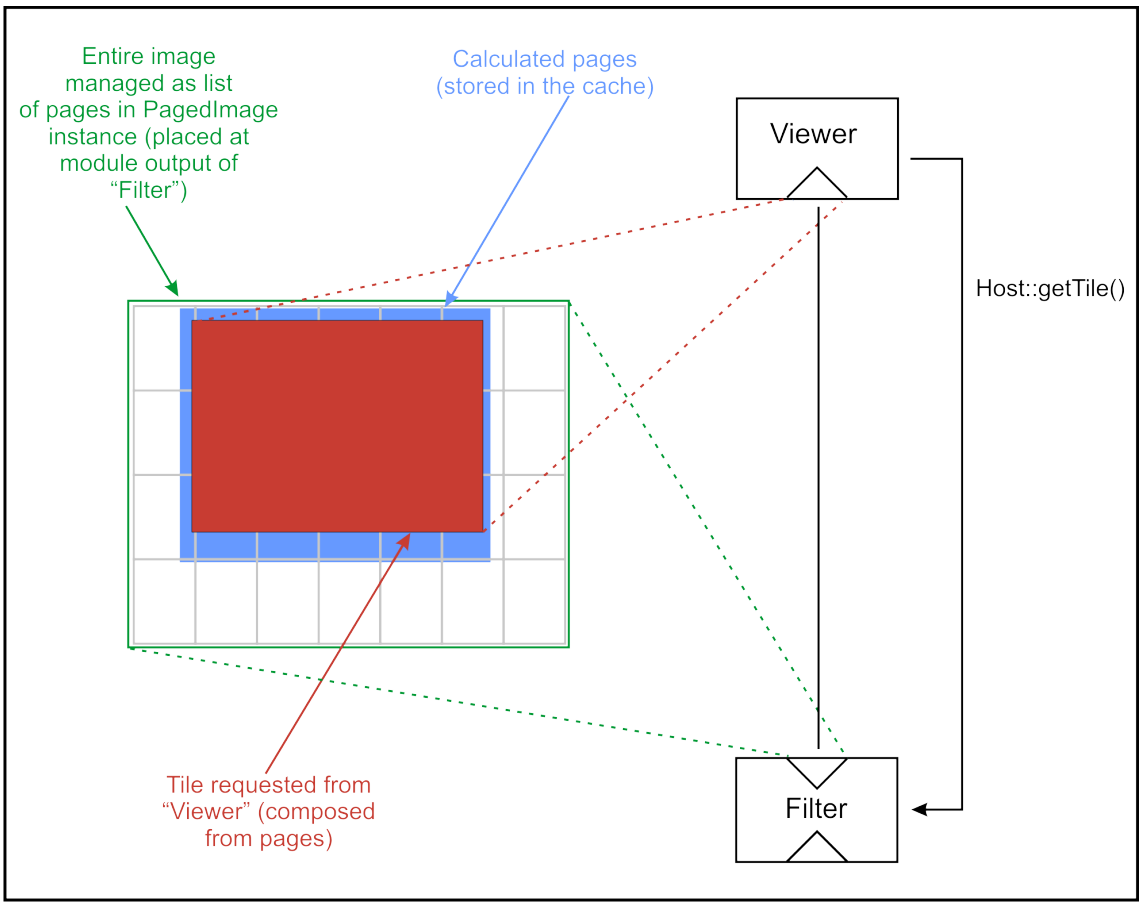
Figure 2.2. Page-Based getTile() (I)



The `Host` calculates a tile with `getTile()` as follows:

- Allocate memory for the tile.
- Calculate the pages that are intersected by the tile.
- For all pages:
  - Already in cache? Yes => Done.
  - Not in cache? => Allocate page in cache.
  - Request to `Filter` with `calculateInputSubImageBox()`: Determine input area needed to calculate the page.
  - Allocate and calculate the input tiles by recursively calling `getTile()`.
- Call `calculateOutputSubImage()` in `Filter`.
- Copy pages to tile.

Figure 2.3. Page-Based getTile (II)



2.2.2. `Memory`

The ML class `Memory` provides functions for memory allocation, reallocation, freeing, etc. Currently, only basic functionality is available; however, future versions will use automatic strategies to (re)organize and/or clean up memory (and the ML cache) to reduce or avoid out-of-memory errors.

Important

If possible, always try to use the memory handling functionality of this class when you need to allocate your own memory.

This class can automatically handle memory errors and will support correct and safe memory handling in the future.

Note

You can use the `mlAPI` functionality instead of the `Memory` class. The `mlAPI` functionality uses - of course - the `Memory` class.

Note

See Section 5.5, “Tracing, Exception Handling and Checked Object Construction/ Destruction” [111] for alternative memory management concepts.

The following class functionality is currently available:

1. `static void* allocateMemory(MLuint size, MLMemoryErrorHandling handleFailure);`

   Allocates a memory block of *`size`* bytes.

2. `static void* reallocateMemory(void* ptr, MLuint size, MLMemoryErrorHandling handleFailure);`

   The memory block pointed to by *`ptr`* is resized and copied so that it has at least *`size`* bytes.

3. `static void freeMemory(void* ptr);`

   Frees memory that has been allocated with any `Memory` function. NULL pointers may be passed safely; they are simply ignored.

4. `static void* duplicateMemory(const void *ptr, MLuint size, MLMemoryErrorHandling handleFailure);`

   Copies the memory pointed to by *`src`* of size *`size`* in a newly allocated buffer which must be freed by the caller with `freeMemory()`. If *`ptr`* is passed as `NULL`, `NULL` is returned without any error handling.

5. `static char* duplicateString(const char *ptr, MLMemoryErrorHandling handleFailure);`

   Copies the passed null-terminated string *`str`* in a newly allocated buffer which must be freed by the caller with `freeMemory()`.

Note

Always use functions of the class `Memory` in ML contexts so that the ML can optimize memory usage and provide safer memory allocations.

The parameter *`handleFailure`* determines the function behavior in error cases:

1. `ML_RETURN_NULL`

   If memory allocation fails, `NULL` is returned without error handling. The programmer must take care of the error.

2. `ML_FATAL_MEMORY_ERROR`

   If memory allocation fails, `ML_PRINT_FATAL_ERROR()` with error code `ML_NO_MEMORY` is called; `NULL` is returned if `ML_PRINT_FATAL_ERROR()` has been returned. The programmer does not need to take care of the error case, because the ML handles it.

3. `ML_THROW_NO_MEMORY`

   If memory allocation fails, `throw(ML_NO_MEMORY)` is executed. The programmer could implement something like

Example 2.4. Using Exceptions when Allocating Memory with `MLThrowNoMemory`

```
ML_START_NAMESPACE

  try {

    // Try to allocate...

    Memory::allocateMemory(1000, ML_THROW_NO_MEMORY);
  }
  catch(MLErrorCode)
  {
    // Handle error if memory could not be allocated.
  }

ML_END_NAMESPACE
```

Note that these error handling cases will only occur if the `Memory` class functionality has no chance to allocate the required memory. In future versions, the following might happen: The first internal allocation fails, but the `Memory` class clears the ML cache and successfully retries memory allocation. In those cases, none of the above error cases will be used.

2.2.3. `Base`

The `ml::Base` class is a base class of many ML classes and is designed for all objects passed between different ML modules via the so-called `Base` fields. Thus it is possible to establish transfer of arbitrary data types between modules.

2.2.4. The Runtime Type System

The ML provides a so-called *Runtime Type System* for managing all important classes available in the module database and in the ML.

- `Runtime`

  This class contains the global runtime type system of the ML. It manages a dictionary of runtime types and can create and remove runtime types. This class only contains static components and must be initialized with `init()` and destroyed with `destroy()`.

- `RuntimeType`

  This class contains runtime-generated type and inheritance information of associated classes. To track this information, the macros defined in `mlRuntimeSubClass.h` have to be inserted in the declaration and implementation of the associated classes.

- `RuntimeDict`

  This class manages a set of instances of the class `RuntimeType`. The class `Runtime` uses one global instance of this class for the runtime type system of the ML.

The file `mlRuntimeSubClass.h` also includes important and frequently used macros.

- `ML_BASE_IS_A(base,type)`

  This macro is used to check whether the given `Base` pointer is of the expected type: `ML_BASE_IS_A(base, MarkerExample)`.

Note

The macro `ML_BASE_IS_A` should be replaced by the explicit cast `mlbase_cast<BaseTpe*>(object)` introduced in the ML version since MeVisLab 2.0.

One of the following macros in the header implementation of a class derived from `Base` must be used.

Each of these macros implements the interface to the runtime type system of the derived class.

- `ML_CLASS_HEADER(className)`

  This macro must be included in the header of a non-abstract class to declare some additional methods described below.

- `ML_MODULE_CLASS_HEADER(ModuleClassName)`

  This macro must be included in the header of a class derived from the class `Module` to declare some additional methods described below.

- `ML_ABSTRACT_CLASS_HEADER(className)`

  This macro must be included in the header of an abstract class to declare some additional methods described below.

One of the following macros in the source file implementation of a class derived from `Base` must be used.

Each of these macros implements the interface to the runtime type system of the derived class.

- `ML_CLASS_SOURCE(className, parentName)`

  This macro must be included in the source file of a non-abstract class to implement the methods declared with `ML_CLASS_HEADER`.

- `ML_MODULE_CLASS_SOURCE(className, parentModule)`

  This macro must be included in the source file of classes derived from the class `Module` to implement the methods declared with `ML_MODULE_CLASS_HEADER`. `Module` implements protected constructors and assignment operators to avoid the assignment of `Module` modules to themselves. The normal `ML_CLASS_SOURCE` macros cannot be used.

- `ML_ABSTRACT_CLASS_SOURCE(className,parentName)`

  This macro must be included in the source file of an abstract class to implement the methods declared with `ML_ABSTRACT_CLASS_HEADER`.

- `ML_MODULE_CLASS_SOURCE_EXT(className, parentModule, superClassConstructs)`

  This macro is an alternative to `ML_MODULE_CLASS_SOURCE` if the constructor of the *`parentModule`* does not have two parameters or if other members need to be initialized (e.g. constants). The third parameter *`superClassConstructs`* permits the specification of the correct constructor call of the superclass, e.g., `ML_MODULE_CLASS_SOURCE_EXT(MyFilter, MyParentModule, :MyParentModule())` does not pass parameters for the superclass constructor that is used in the normal `Module`.

  If you need to pass more complex expressions as third parameters, such as superclass or member initializers (as a comma-separated list, for example), use the following trick:

Example 2.5. How to Use the Macro `ML_MODULE_CLASS_SOURCE_EXT`

```
// Stuff to do for base classes when copy constructor is implemented
// (which is done in a macro to have a private and not an executable
// copy constructor).

#define _INIT_STUFF : Module(0,0), _initMember1(initValue1), \
                                   _initMember2(initValue2)


// This macro declares some automatically generated functions and methods
// for the runtime system and for the initialization of this class. It
// implements more elaborated superclass and member initializers given
// by _INIT_STUFF.

ML_MODULE_CLASS_SOURCE_EXT(MyNewModule, Module, _INIT_STUFF)

#undef _INIT_STUFF
```

See also the file `mlLibraryInitMacros.h` which does not directly belong to the runtime type system but which contains macros for the initialization after runtime linking to the library. It permits the implementation of a function in a library where module classes and runtime types can be initialized directly after linking to the library.

2.2.5. Debugging and Error Handling Support

See Chapter 5, *Debugging and Error Handling* for detailed information on concept, classes and macros for error handling and debugging.

2.3. Image Classes

2.3.1. `ImageProperties`

The `ml::ImageProperties` class describes the basic image properties

- 6 dimensional extent as a ImageVector (Section 2.4.1, "`ImageVector`, ImageVector"),
- the voxel data type (`MLDataType`),
- the minimum and maximum limits of voxel values, and

Images are rectangular grids without gaps, and all voxels are of identical extent and types. The six image dimensions in the ML are interpreted as the three spatial dimensions (x, y and z), a color extent (c dimension), a time extent (t dimension) and a user (u) dimension. For example, a dynamic sequence of three dimensional color images that exist in different image acquisitions or reconstructions can be handled by the ML as a single image object.

See `mlImageProperties.h` in project ML for more information.

2.3.2. `MedicalImageProperties`

The `ml::MedicalImageProperties` class is derived from `ImageProperties` (Section 2.3.1, "`ImageProperties`"). It contains additional information specialized for medical data sets:

- a voxel size,
- a 4x4 transformation matrix (world matrix) to specify 3D transformations (e.g., for registration purposes),
- an anonymous reference to a list that stores DICOM tag information if the input file(s) have been in DICOM file format,

- color channel information (as strings) for the 4th dimension. The string list `std::vector<std::string> &getCDimensionInfos()` describes the significance for the channels e.g., "RED", "GREEN", and "BLUE" for channels 0, 1 and 2 when the RGB color model is used,
- time point information for the t extent of the image. The list `std::vector<DateTime> &getTDimensionInfos()` contains this information for each time point,
- u dimension information given as a list accessible with `std::vector<std::string> &getUDimensionInfos()`. The stored strings describe the subimages with different u components. Often, strings such as "CT", "MR", etc. are stored.

Note

For the c (color) and u dimension, there is a set of constants available describing the image contents, such as `ML_RED`, `ML_BLUE`, `ML_SATURATION`, `ML_HUE` for the c dimension, or `ML_CT`, `ML_MR`, `ML_PET` for the u dimension (see `mlDataTypes.h`). The components of the list for the t dimension are given by the class `DateTime` (see `mlDateTime.h`).

See `mlImageProperties.h` in project ML for more information.

2.3.3. `ImagePropertyExtension`

`ml::ImagePropertyExtension` is used to append additional and user-defined property information to an ML image. This class is independent of the classes `ImageProperties` and `MedicalImageProperties` (see Section 2.3.1, “ `ImageProperties` ” and Section 2.3.2, “ `MedicalImageProperties` ”). It is an abstract class that serves as a base class from which an application or programmer can derive new properties. These properties are added to the `ImagePropertyExtensionContainer` that is a member of the class `MedicalProperties`.

A derived ImagePropertyExtension must meet some requirements:

- It must implement the copy constructor and assignment operator correctly, because objects of its type are copied from one image to another.
- It requires a virtual `createClone()` method that returns a copy or a new instance of the class so that a copy of the correct derived class is returned.
- It must implement ML runtime typing and must be registered in the runtime type system of the ML in such a way that the ML can create new instances of the user-defined class only from its name and compare class types.
- It must implement set and get methods to set/get the property value as a string, because ML modules must be able to store/load property settings in/from a file.
- It must implement equality and inequality operators to compare instances.

Most methods to be implemented are pure virtual in the base class `ImagePropertyExtension`, hence compilation will not work without implementing them

The following programming example demonstrates how to implement a newly derived `ImagePropertyExtension`:

```
#include "mlModuleIncludes.h"

ML_START_NAMESPACE

//! Implement a ImagePropertyExtension object which can be passed to the ML.
class MODULE_TESTS_EXPORT OwnImagePropertyExtension : public ImagePropertyExtension
{

public:

  //! Constructor.
```

```
  OwnImagePropertyExtension() : ImagePropertyExtension()
  {
    _extInfoString = "NewImageInfosString";
  }

  //! Destructor.
  virtual ~OwnImagePropertyExtension() { }

  //! Implement correct copy construction.
  OwnImagePropertyExtension(const OwnImagePropertyExtension &origObj) :
    ImagePropertyExtension(origObj)
  {
    _extInfoString = origObj._extInfoString;
  }

  //! Implement correct assignment.
  OwnImagePropertyExtension &operator=(const OwnImagePropertyExtension &origObj)
  {
    if (&origObj != this) { _extInfoString = origObj._extInfoString; }
    return *this;
  }

  //! Implement pure virtual equality operation to work even on base class pointers.
  virtual bool equals(const ImagePropertyExtension &extImageProps) const
  {
    if (extImageProps.getTypeId() == getClassTypeId()) {

      // Types are equal, compare contents.
      return _extInfoString == ((OwnImagePropertyExtension&)(extImageProps))._extInfoString;

    } else {
      return false; // Types differ, thus objects also differ.
    }
  }

  //! Creates a copy of the correct derived object (for comparisons / runtime type determination).
  virtual ImagePropertyExtension *createClone() const
  {
    return new OwnImagePropertyExtension(*this);
  }

  //! Returns value of property as string.
  virtual std::string getValueAsString() const
  {
    return _extInfoString;
  }

  //! Set value of property from string value.
  virtual MLErrorCode setValueFromString(const std::string &str)
  {
    _extInfoString = str;
    return ML_RESULT_OK;
  }


private:

  //! The string values used as additional image property.
  std::string _extInfoString;

  //! Implements interface for the runtime type system of the ML.
  ML_CLASS_HEADER(OwnImagePropertyExtension)
};

ML_END_NAMESPACE
```

Implement the C++ part of the class interface to the runtime type system:

```
ML_START_NAMESPACE

  //! Implements code for the runtime type system of the ML.
  ML_CLASS_SOURCE(OwnImagePropertyExtension, ImagePropertyExtension);

ML_END_NAMESPACE
```

Register the class to the runtime type system of the ML when the .dll/.so file is loaded. This is typically done in the InitDll file of the project:

```
ML_START_NAMESPACE

  int MyProjectInit()
  {
```

```
    OwnImagePropertyExtension::initClass();
  }

ML_END_NAMESPACE
```

In the method calculateOutputImageProperties of your ML module, you can add a copy of your own image property to the output image:

```
ML_START_NAMESPACE

  MyModule::calculateOutputImageProperties(int outIndex, PagedImage* outImage)
  {
    OwnImagePropertyExtension myNewImgProp;
    outImage->getImagePropertyContainer().appendEntry(&myNewImgProp, true);
  }

ML_END_NAMESPACE
```

See `mlImagePropertyExtension.h` and `mlImagePropertyExtensionContainer.h` in project ML for more information.

2.3.4. `PagedImage`

The class `ml::PagedImage` is dedicated to managing paged images in the ML and to representing image outputs of ML modules. See `mlPagedImage.h` in project ML.

The ML mainly works with pages and tiles. Since ML does usually not process entire images, it is necessary to break them down into smaller fractions of identical extent, the so-called *pages*. Pages can easily be buffered, cached and processed in parallel without spending too much memory or time. Caching in the ML works exclusively with pages. Moreover, only the pages that overlap with the actually requested image region must be processed. All other pages are not processed. Common page extents are, for example, 128x128x1x1x1x1 voxels. However, they may also have a real six-dimensional extent as do all images in the ML. Often, other image fractions (also called *tiles*) which do not have standard page extents are needed. Tiles are usually composed from pages and are used by the application or as input for image processing algorithms. In the ML, tiles are usually only temporary, i.e., the ML does not cache tiles.

For algorithms where a page-based implementation is difficult, classes such as `VirtualVolume`, `BitImage` or `MemoryImage` provide special interfaces to simplify efficient implementations. See Section 2.3.7, “ `VirtualVolume` ”, Section 2.3.6, “ `BitImage` ” and Section 2.3.8, “ `MemoryImage` ” for more information.

2.3.5. `SubImage/TSubImage`

`ml::SubImage` is an important class representing image and subimage buffers. It is used to manage, copy, etc. chunks of voxel data. It contains fast data access methods. See `mlSubImage.h` .

`ml::TSubImage` is the typed version of `SubImage` which permits typed data accesses. See `mlTSubImage.h` in project ML.

`ml::TSubImageCursor` and `ml::ConstTSubImageCursor` are cursor classes that allow access to a given TSubImage using cursor positioning and movement.

The `SubImage` class represents a rectangular 6D image region with linearly organized memory. It offers:

- Methods for setting/accessing the datatype, the box defining the subimage region, the source image extent and the valid region (which is the intersection of the source image extent and the box).
- A pointer to the memory data containing the image data as a void pointer. Alternatively the data can be stored as a `MLMemoryBlockHandle` to manage data via the `MLMemoryManager`.
- With this class, the `Host` manages and encapsulates rectangular image regions (e.g., for pages, tiles, cached image results) and passes them to the image processing algorithms. The `Host` usually does not need information about the actual data.

- The typical image processing methods in the ML are located in overloaded methods of `Module::calculateOutputSubImage()`. In these methods, the untyped memory chunks given as `SubImage` are usually wrapped again to typed subimages. See `TSubImage` for more information.
- The type of the image data in memory is handled via a void pointer; the type, however, is managed as an `enum` type to support typed access in derived classes (see `TSubImage`). Consequently, `SubImage` does not support typed access to image voxels.
- The typed access to voxels is implemented on top of this class as the template class `TSubImage`.

2.3.5.1. Example

The following paragraphs show some typical use cases of the class `SubImage`.

This creates a `SubImage` instance that provides access to a chunk of double data of 16 x 32 x 8 x 1 x 1 x 1 voxels given by the pointer *`dataPtr`*:

```
SubImage subImgBuf(SubImageBox(ImageVector(0,0,0,0,0,0),
                               ImageVector(15,31,7,0,0,0)),
                   MLdoubleType,
                   dataPtr);
```

The caller is responsible for the data chunk to be sufficiently large.

This first fills the entire subimage with the value `7.7`. Then the rectangular region outside the area given by (3,3,3,0,0,0) and (5,5,5,0,0,0) is filled with the value `19.3`:

```
subImgBuf.fill(7.7);
subImgBuf.fillBordersWithLDoubleValue(SubImageBox(ImageVector(3,3,3,0,0,0),
                                                  ImageVector(5,5,5,0,0,0)),
                                      19.3);
```

Assuming another `SubImage` object *`srcSubImg`*, the overlapping areas can simply be copied (and if necessary, cast to the target type) into *`subImgBuf`*, and optionally rescaled with value 0.5:

```
subImgBuf.copySubImage(srcSubImg, ScaleShiftData(0.5, 0) );
```

Untyped data access to the voxel data is available for example at position (1,2,3,0,0,0) with

```
void *voxPtr = subImgBuf.getImagePointer( ImageVector(1,2,3,0,0,0,0) );
```

For typed data management, the class `TSubImage` can be used almost in the same way:

```
TSubImage<MLdouble> subImgBufT(SubImageBox(ImageVector(0,0,0,0,0,0),
                                           ImageVector(15,31,7,0,0,0)),
                               MLdoubleType,
                               dataPtr);
```

The class `TSubImage`, however, provides a number of typed access functions, such as

```
MLdouble *voxPtrT = subImgBufT.getImagePointer( ImageVector(1,2,3,0,0,0,0) );
*voxPtrT = 29.2;
```

The untyped `SubImage` and the templated `TSubImage` classes also provide a variety of other methods to manipulate, copy, fill, allocate and delete data, or to check for a certain value, or to retrieve statistical information such as minimum or maximum. They are powerful classes that can be used in many contexts when memory or voxel buffers have to be managed.

See files `mlSubImage.h` and `mlTSubImage.h` for more information.

2.3.6. `BitImage`

In the page-based image processing concept of the ML, Boolean data types are not available (nor are they planned).

The `BitImage` class can be used as an alternative.

The following set of operations is available for this class type:

- full 6D support in all methods,
- set, get, clear and toggle bits at coordinates,
- filling (=clearing or setting) and inverting subimage boxes,
- copying from/to subimages (with thresholding),
- saving/loading to/from file,
- position checking,
- creating downscaled `BitImage`s,
- creating `BitImage`s from image data where first the mask area is determined and then the smallest possible `BitImage` is returned,
- cursor movement in all dimensions,
- exception handling support for safe operations on images.

2.3.7. `VirtualVolume`

The `ml::VirtualVolume` and the `ml::TVirtualVolume` classes manage efficient voxel access to the output image of an input module or to a 'standalone' image.

So it is possible to implement random access to a paged input image or to a pure virtual image without mapping more than a limited number of bytes. Pages of the input volume are mapped temporarily into memory when needed. If no input volume is specified, the pages are created and filled with a fill value. When the permitted memory size is exceeded, older mapped pages are removed. When pages are written, they are mapped until the virtual volume instance is removed or until they are explicitly cleared by the application. Virtual volumes can easily be accessed by using `setValue` and `getValue`. These kinds of access are well-optimized code that might need 9 (1D), 18 (3D) and 36 (6D) instructions per voxel if the page at the position is already mapped.

A cursor manager for moving the cursor with `moveCursor*` (forward) and `reverseMoveCursor*` (backward) is also available. `setCursorValue` and `getCursorValue` provide voxel access. Good compilers and already mapped pages might require about 5-7 instructions. So the cursor approach will probably be faster for data volumes with more than 2 dimensions.

All the virtual volume access calls can be executed with or without error handling (see last and default constructor parameters). If *`areExceptionsOn`* is `true`, every access to the virtual volume is tested and if necessary, exceptions are thrown that can be caught by the code calling the virtual volume methods. Otherwise, most functions do not perform error handling.

Note

Exception handling versions are slower than versions with disabled exceptions. However, this is the only way to handle accesses safely.

Tip

This class is the recommended alternative to global image processing algorithms.

2.3.7.1. Code Examples

The following code gives an example of how to use the `VirtualVolume` class:

Example 2.6. How to Use the `VirtualVolume` Class

Header:

```
VirtualVolume *_virtVol;
```

Constructor:

```
_virtVol = NULL;
```

Create/Update the virtual volume in `calculateOutputImageProperties()` and invalidate the output image on errors, so that `calculateOutputSubImage`() is not called on bad virtual volume later.

```
if (_virtVolume != NULL) { delete _virtVolume; }

_virtVolume = new VirtualVolume(this, 0, getInputImage(0)->getDataType());

if (!_virtVolume || (_virtVolume && !_virtVolume->isValid())){
  outImage->setInvalid(); return;
```

When you do not want to use a 'standalone' virtual volume:

```
_virtVolume = new VirtualVolume(ImageVector(1024,1024,1,1,1,1), 0, MLuint8Type));

if ((_virtVolume == NULL) || (_virtVolume && !_virtVolume->isValid())){
  outImage->setInvalid(); return;
}
```

Example of how to access image data directly: `calculateOutputSubImage()`

```
// Create wrapper for typed voxel access.
TVirtualVolume<DATATYPE> vVol(*_virtVolume);

ImageVector pos(7,3,0,0,0,0);
DATATYPE value;

vVol.setValue(pos, value);             // Simple setting of an arbitrary voxel.
value = vVol.getValue(pos);            // Reading of an arbitrary voxel.
vVol.fill(outSubImg->getBox(), value); // Fill region with value.

// Now copy valid region of virtVolume to outSubimg.
vVol.copySubImage(*outSubImg);
```

Example of how to access image data via a cursor: `calculateOutputSubImage()`:

```
// Create wrapper for typed voxel access.
TVirtualVolume<DATATYPE> vVol(*_virtVolume);

ImageVector pos(7,3,0,0,0,0);
DATATYPE value;

vVol.setCursorPosition(pos);               // Set cursor to any position in volume.
vVol.moveCursorX();                        // Move cursor >F<orward
vVol.moveCursorY();                        // in (positive) X, C and U direction.
vVol.moveCursorZ();
vVol.reverseMoveCursorT();                        // Move cursor >B<ackwards in (negative) T
vVol.reverseMoveCursorZ();                        // and Z direction.

val = vVol.getCursorValue();           // Reading voxel below cursor.
vVol.setCursorValue(10);               // Set voxel value below cursor to 10.
```

Additionally, the following helper routines are available:

```
// Fill region of virtual volume with a certain value.

void fill(const SubImageBox &box, DATATYPE value);


// Copy region from the virtual volume into a typed subimg.

void copyToSubImage(TSubImage<DATATYPE> &outSubImg);
```

```
// Copy a region from a typed subimg into the virtual volume.

void copyFromSubImage(TSubImage<DATATYPE> &inImg,
                      const SubImageBox &box,
                      const ImageVector &pos);
```

There are also some routines to get the boxes of the currently written pages. It is also possible to read/write the data of the written pages directly.

Note

The class `VirtualVolume` contains data structures for data management and table caching; its creation is expensive in comparison to the `TVirtualVolume` class which is only a "lightweight" access interface that can rapidly be created and destroyed on top of a `VirtualVolume` object. In the case of algorithms which implement template support for arbitrary data types it is recommended to create an untyped `VirtualVolume` as class member and a `TVirtualVolume` class in `calculateOutputSubImage` for maximum performance.

However, there are also convenience constructors of the `TVirtualVolume` class which internally create the `VirtualVolume` instance automatically ; these constructors are more expensive and should not be used on each **calculateOutputSubImage** call. Nevertheless they can be useful when a class works only with a fixed data type or without templates.

Using virtual Volume instances that create untyped virtual volume instances automatically.

Creating a `TVirtualVolume` with a convenience constructor. It creates a `VirtualVolume` internally. It provides float data access to the input image 0, even if the input image is of another type. Note that the connected input image must be valid:

```
// Create a typed VirtualVolume from input connector 0 of
// this Module with voxels of type float directly without
// creating the untyped VirtualVolume manually.

TVirtualVolume<float> vVol(this, 0);
```

2.3.7.2. Using Exceptions for Safe `VirtualVolume` Usage

The standard usage of the `VirtualVolume` and the `TVirtualVolume` classes does not include error handling. For safe usage, *`areExceptionsOn`* == `true` is passed as a parameter to the constructor, and errors will throw the following exceptions:

Note that *`areExceptionsOn`* == `true` degrades voxel access performance.

- `ML_OUT_OF_RANGE`

  `ML Error Code`

  `MLErrorCode` is thrown if cursor positioning or voxel addressing tries to access invalid image regions. The exception leaves the virtual volume, the cursor position, the voxel content, etc. unchanged and the invalid flag of the virtual volume is **not** set. The call is just terminated and ignored, i.e., The call can continue and accesses to other voxels are attempted.

- `ML_NO_MEMORY`

  `MLErrorCode` is thrown if an allocation fails because of insufficient memory. The valid virtual volume is invalidated, i.e., Its valid flag is cleared.

- `ML_BAD_DIMENSION`

  `MLErrorCode` is thrown if the image data extent is invalid. This could indicate a programming error or invalid input image data. The valid virtual volume is invalidated, i.e., Its valid flag is cleared.

- `ML_BAD_DATA_TYPE`

  `MLErrorCode` is thrown if an invalid image data type is encountered. This could indicate a programming error or invalid input image data. The valid virtual volume is invalidated, i.e., Its valid flag is cleared.

- Other exceptions that result from page request failures could also be thrown. They are usually returned, when a `getTile` command that attempts to request data from an input image fails.

If the *`areExceptionsOn`* == `false`, no exception is thrown and many errors are handled by calling the ML_PRINT*() error macros and terminating the function/method. The virtual volume instance will be invalidated. Invalid voxel access or memory failures will destroy the program state or cause unknown exceptions.

2.3.7.3. Performance Issues on `VirtualVolume` Usage

- Voxel access performance is best when the page extents of input pages are powers of 2.
- Working locally on virtual volumes is generally faster than jumping randomly through the image, because less pages must be swapped.
- Coordinate-specific voxel access performance is better for images of a lower dimension, because less calculations have to be performed.
- If the virtual volume wraps a paged input image, voxel access is not permitted when the input connection or the module has become invalid.
- The virtual volume must not be used in parallel in `calculateOutputSubImage()` calls, because `getValue` and `setValue` methods potentially call `getTile*()` which would start recursive multithreading. Therefore be sure that multithreading remains disabled in areas where `VirtualVolume` or `TVirtualVolume` use `calculateOutputSubImage()` or you must make accesses to them thread-safe by using critical sections, semaphores or similar concepts. Even if no paged image is used as an input, write access is not capable of multithreading due to performance reasons.
- If an image has n dimensions (e.g., 3), components >= n in cursor positioning and voxel access are simply ignored for performance reasons and do not cause errors if they are set even if this means that the cursor was outside the image.
- In some cases, the virtual volume approach is slower than a global approach.

  Consider the following reasons:

  - The virtual volume approach is completely page-based, i.e., it fits perfectly in the optimized page-based concept of the ML.
  - The virtual volume approach only requests image areas of the input image that are really needed (processing on demand) so that less input image regions are calculated. Global approaches always request the entire input image which is often expensive to calculate.
  - The virtual volume approach usually locks less memory than the global approach, so the operating system must swap less memory, and other modules can work faster.
  - Next versions will not duplicate memory as their own tiles (as a global approach needs to), but will directly try to use ML cache pages.

2.3.8. `MemoryImage`

`MemoryImage` can be used for algorithms that need fast random access to entire images, especially if they work “against“ paging e.g., `OrthoReformat`, `MPR`, `MemCache`.

Important

- A `MemoryImage` object is always buffered at the output of the connected input module.
- Try to avoid this approach! It only supports limited image sizes that depend on the available memory! See Section 4.3.3, “`VirtualVolume` Concept” for information on how to avoid this concept.

Properties:

- The `MemoryImage` object is part of each `PagedImage`, i.e., there is one (usually empty and unused) `MemoryImage` object per output.
- If possible, all connected ML modules copy or reference data directly from the `MemoryImage` object.

There are two ways of how to use the memory image at a module output:

- The module completely controls the `MemoryImage` object at the image output (reset, clear, set, resize, update...). Thus connected modules benefit (see Version 1).
- The entire input image is requested as one page (with the note to buffer it as a memory image). Further requests (also from other modules) will be answered immediately by passing the pointer to the memory image or by copying page data from it (see Version 2).

- Version 1: The module controls the memory image at the output:

Example 2.7. Controlling the `MemoryImage` by the Module

```
// Constructor: Enables the operator control of the memory output at output 0.

getOutputImage(0)->getMemoryImage().setUserControlled(true);


// Resize and copy input image into the memory image output:

MLErrorCode result = getOutputImage(0)->getMemoryImage().update(
        getInputImage(0),
        getInputImage(0)->getImageExtent(),
        getInputImage(0)->getDataType());
if (ML_RESULT_OK != result) { handleErrorHere(result); }


// Get data pointer and draw into memory image at output:

drawSomethingsIntoImg(getOutputImage(0)->getMemoryImage().getImage());
```

- Version 2: The memory image is cached at the output of preceding module:

Example 2.8. Using/Requesting a `MemoryImage` of the Predecessor Module

```
// Request input tile caching in output of input module
void MemoryInTest::calculateOutputImageProperties(int outIndex, PagedImage* outImage)
{
  ...

  outImage->setInputSubImageUseMemoryImage(0, true);

}

// Request input tile of size of input volume (other sizes cause warnings!)

SubImageBox MemoryInTest::calculateInputSubImageBox (int /*inIndex*/,
                                          const SubImageBox & /*outSubImgBox*/,
                                          int /*outIndex*/)
{
  return getInputImage(0)->getBoxFromImageExtent();
}
```

Advantages:

- All connected modules can benefit from the memory image, because it is part of the image output.
- It is easy to implement and fast; it does not break the paging concept.

Disadvantages:

- The image size is limited by the size of the largest free memory chunk.
- It cannot/should not be used in bigger networks or applications.
- It must map the entire image and blocks large memory areas for a long time.

2.4. Helper Classes

2.4.1. `ImageVector`, ImageVector

The `ml::ImageVector` class manages a 6D point/vector with 6 components *x*, *y*, *z*, *c*, *t*, *u* of the type `MLint`. It is the main class used for voxel positions, image extents, box corners or page extents. The typical (integer) vector arithmetics is available as well as minimum and maximum component search, lexicographical comparison, stride operations and component-wise multiplication for voxel addressing, etc. It offers different template specializations with 16, 32 and 64 bit integer components, because most ML image and voxel addressing is done by this class. A `ImageVector` works with 64 bit integers to support very large image addressing and should be used in all image processing modules unless there are clear reasons for using specialized versions. The most important methods are:

- operators +, -, *, /, <<, >>, []

  Operators as they are defined on integers, applied component-wise.

- `getStrides()`

  If a `ImageVector` is interpreted as an image extent, this method returns another `ImageVector` with voxel offsets (usually called *strides* in such a context). To get from one voxel position to a neighbor position, e.g., to y, the corresponding stride `getStrides().y` has to be added. The y strides are usually identical with the x extent of the image, the z stride with the number of x extent multiplied by the y extent, the c stride with x extent*y extent*z extent , etc. Strides are typically needed for very fast voxel positioning with indices into images.

- `getVectorPosition(offsetPos)`

  If the `ImageVector` is interpreted as the extent of an image and *`offsetPos`* as an index into the image, `getVectorPosition()` returns the position of the voxel as `ImageVector`.

- `getExtDimension()`

  If the `ImageVector` is interpreted as the extent of an image, this method returns the highest `ImageVector` component that cannot be used to get the real dimension of an image.

- `hasNegativeComp()`

  This method returns `true` if any component is negative, otherwise it returns `false`. This method can be used to check whether the position might be an invalid image coordinate (negative components are not used for voxel addressing).

- `allBiggerZero()`

  This method returns `true` if all components > 0, otherwise it returns `false`.

- Constructors, set and get methods to initialize the `ImageVector` in different ways, maximum/minimum component search, etc. are available.

Please refer to file `????` in the project *MLLinearAlgebra* for more information (Section 2.6.1, "`MLLinearAlgebra`(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)").

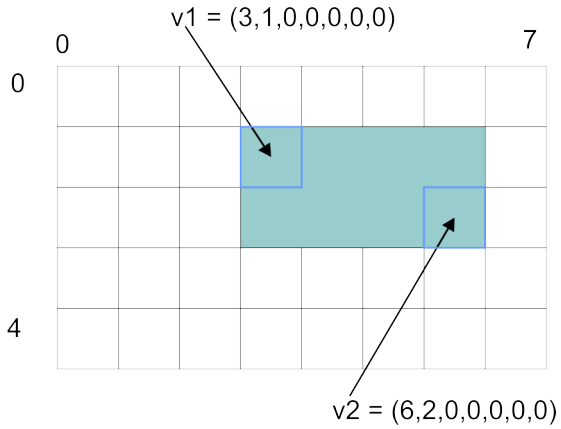
2.4.2. `SubImageBox`

The `ml::SubImageBox` class manages a rectangular 6D box given by two corners that are specified by the `Vectors` *v1* and *v2*. It permits volume intersection, calculation, etc. `SubImageBox` is available in 16, 32 and 64 bit template specializations (like the class `ImageVector`). Use `SubImageBox` without a number for all normal ML code. See `mlSubImageBox.h` in the project ML. A comparable class with 6D float vector is available as `SubImageBoxd` in file `mlSubImageBoxd.h` .

Important

- Both corners of the box are considered to be *inside* the box, so a `SubImageBox` from (13,12,10,0,0,0) to (13,12,10,0,0,0) contains exactly one voxel.
- The SubImageBox is empty if any of the *v1* components is bigger than the corresponding *v2* component.

Figure 2.4. SubImageBox



This class offers a set of useful methods, e.g.:

- `getSize()`

  Returns the number of voxels in the subimage region, i.e. the product of all extents if this is not empty; otherwise 0 is returned.

- `isEmpty()`

  Returns `true` if the box is empty; otherwise `false` is returned.

- `intersect(loc2)`

  Returns the regions that overlap with subimage regions *loc2* as a `SubImageBox`. In case of non-overlapping boxes, the returned box is empty.

- `contains (pos)`

  Returns `true` if pos is within box; otherwise `false` is returned.

- `getExtent()`

Returns a vector with the extent of the box in all 6 dimensions (see Section 2.4.1, “ImageVector, ImageVector”).

- `translate(shift)`

  Translates the box by the vector shift.

- `get3DCorners(...)`

  Returns all eight corners of the box as vectors (see Section 2.4.1, “ImageVector, ImageVector”).

Please refer to file `mlSubImageBox.h` for more information.

2.5. APIs and Classes for Interfaces and Voxel Type Extensions

2.5.1. How Applications and the ML Work

An easy way to use the ML is just to link the C-API (Section 6.3, “mlAPI.h”) of the ML. Functions are available to create and delete modules, to set and get their parameters (fields), to connect them and to request images from their outputs.

Example:

Requesting image data from a module causes the following (as shown in Fig. 2.4):

- The `Host` starts to determine the image areas needed by `Viewer` and breaks the area down into pages.
- For each page, the `Host` determines the input data the `Filter` needs for output calculation.
- The `Loader` requests the data (as a set of pages) and composes the correct input data for the `Filter`.
- The `Filter` is called to calculate the output page.
- When `Filter` has calculated the correct output pages, they are composed to the correct image data to be used by the `Viewer`.
- All pages of all modules are stored in the cache of the `MLMemoryManager` if there is enough space.

2.5.2. The C-API

The C-API (C-***A****pplication* ***P****rogramming* ***I****nterface*) is an interface to most C++ functionality of the ML which can be linked in standard C mode. Thus programs and applications not written in C or C++ can link and use the ML if they support standard C linkage. Also, the C-API is more stable, because it is less frequently modified than the C++ interface.

2.5.3. Registering and Using Self-Defined Data Types

The ML supports a set of standard data types for image voxels (8,16,32,64 bit integer types and float, double) (see Section 6.5, “mlTypeDefs.h”) as well as so-called extended data types (see Chapter 7, *Registered Voxel Data Types* and Section 7.5.5, “Implementing a New Voxel Data Type by Deriving from MLTypeInfos”) which permit the usage of self-defined or augmented data types. It is not necessary to recompile the ML for these data types, but modules might need to be adapted depending on how they were written. A structure describing the data type, its properties, and operations can be registered in the ML to activate a new type.

2.6. Tools

The following sections describe classes and toolboxes, and contain a lot of useful information for advanced ML programming.

2.6.1. `MLLinearAlgebra`(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)

This project contains some basic classes for simple vector and matrix arithmetics as well as for quaternion support.

2.6.2. `MLUtilities`

This project contains some basic classes, files and interfaces used by the ML which are not directly related to image processing. Some of them are explicitly explained in other chapters of this document.

1. `DateTime`

   Defines the class `DateTime` for storing and processing date and time values.

2. `mlErrorMacros.h`

   This file contains a number of macros for error handling, checking and tracing that should be used in all ML code. These macros are essential for source code quality control and for many other checks as well as for error tracing and reporting, and for exception handling.

3. `mlErrorOutput` and `mlErrorOutputInfos`

   The class `ErrorOutput` is the main error handling and redirecting class of the ML. It uses `mlErrorOutputInfos` as an information container for error, debug or trace information that is passed to registered error handling routines. See Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs” for more information.

4. `mlFileSystem.h`

   Defines a set of C functions for system independent file management (file open, close, etc.). All methods support UTF-8 unicode strings to access files that contain unicode characters in their absolute file names.

   See also Chapter 9, *Unicode Support* for more information on internationalization and management of files that contain international characters in their file names.

5. `mlLibraryInitMacros.h`

   This file defines macros that are used for platform-independent library initializations with correct version checks. Most importantly, the macro `ML_INIT_LIBRARY(initMethod)` is defined in this file. This macro is used to initialize shared libraries independently of the underlying system (WIN32/ Mac OS X/Linux). When the library has been loaded, the given init method is called as soon as possible. The name of the *`initMethod`* should be composed like < `dllName` > + '`Init`'. This is necessary since this macro sets the name of the initialized DLL as well. Subsequent runtime types will use this name to register the originating DLL.

   See also Section 2.2.4, “The Runtime Type System” for macros to initialize module classes and runtime types of the ML.

6. `mlMemory.h`

   Basic memory management class for the ML.

7. `mlNotify.h`

   Class to notify registered instances of ML changes. With this class, registered classes will be notified of changes to ML internals. It is not intended to be used in normal modules, but can be very useful for diagnostics.

8. `mlRuntime.h` , `mlRuntimeDict.h` , `mlRuntimeSubClass.h` , `mlRuntimeType.h`

   See Section 2.2.4, “The Runtime Type System” for more information on these classes and files.

9. `mlSystemIncludes.h`

   This file includes many important system files, makes correct adaptations for some platforms and disables boring and unproductive warnings. It is designed to be independent of the ML or `MLUtilities` and does not need to link to any ML or `MLUtilities` binary. When this file is used, the most important program parts are provided platform-independently and without any warnings. Include this file instead of directly including system files.

10. `mlTypeDefs.h`

    Header file that contains the most important ML types, constants and definitions; this file can be included without having to link the ML, `MLUtilities` or `MLLinearAlgebra` project (see Section 2.6.1, “`MLLinearAlgebra`(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)”) and it also does not use any C++ functionality. Thus much ML stuff can be used without being really dependent on the ML.

11. `mlUnicode.h`

    File that contains a set of C functions for converting and managing normal unicode strings.

    See also Chapter 9, *Unicode Support* for more information on internationalization and management of files that contain international characters in their file names.

12. `mlVersion.h`

    Public header file for ML version support (Section A.7, “Version Control”). It provides some support for checking for the correct binary versions of the ML and the ML C-API.

2.6.3. Other Classes

2.6.3.1. `SubImageBoxd`

Like `SubImageBox` , only with `Vector6` corners. Manages a rectangular 6D box given by two Vector6. Permits intersections etc. See `mlSubImgBoxf` in project ML.

2.6.3.2. Other Classes and Types

The ML includes some frequently used types.

A detailed explanation of the following helper classes will be given in later editions of this document. Please refer to later versions of this document or to the header files:

- `Engine`

  A class derived from `Module` and intended to build modules similar to Open Inventor™ engines by the use of ML fields. It does not provide any image input or output, i.e., only operations on fields can be implemented.

- `Plane`

Manages a geometric plane in 3D. Used as an encapsulated data type for a `PlaneField`.

- `Rotation`

  Manages a geometric rotation in 3D. Used as an encapsulated data type for a `RotationField`.

- `Disc`

  Manages a geometric 2D disk of a certain radius that is placed in 3D and that can also be voxelized into any image.

- `Sphere`

  Manages a geometric sphere in 3D.

2.6.3.2.1. MLDataType

The `MLDataType` is an enumerator describing all voxel data types currently available in the ML. This includes built-in data types like `MLint8Type`, `MLuint8Type`, `MLint16Type`, `MLuint16Type`, `MLint32Type`, `MLuint32Type`, `MLint64Type`, `MLuint64Type`, `MLfloatType` and `MLdoubleType` as well as (pre)registered types like `Vector2`, `Vector3`, `Vector4`, `Vector6`, `Vector8`, `Vector10`, `Vector16`, `Vector32`, `complexf`, etc.

Values of this type are often used to request or to determine image data or voxels of a certain type.

Since the number of `MLDataTypes` can change during runtime, it is implemented as an integer, and the function `MLNumDataTypes()` returns the current number of available voxel data types. `MLDataTypeNames()` returns a pointer to the table of data type names corresponding to the `MLDataType` values.

Note

- An `MLDataType` value may also be invalid or out of range. This should be taken into consideration when using such values. `MLIsValidType(MLDataType)` can be used to check a type's validity.
- The number of data types may grow during runtime. `MLNumDataTypes()` can be used to retrieve the current number.

A number of other useful functions is available to query information about `MLDataTypes`:

1. `const char *MLNameFromDataType(MLDataType dt)`

   Returns the C string name of data type *`dt`* if *`dt`* is a valid type. Otherwise "" is returned.

2. `MLDataType MLDataTypeFromName(const char * const name)`

   Returns the `MLDataType` value of the data type with the name *`name`*. If *`name`* is not valid, -1 is returned.

3. `double MLDataTypeMax(MLDataType dt)`

   Returns the maximum value of data type *`dt`*; if *`dt`* is invalid, 0 is returned.

4. `double MLDataTypeMin(MLDataType dt)`

   Returns the minimum value of data type *`dt`* if *`dt`* is invalid, 0 is returned.

5. `size_t MLSizeOf(MLDataType dt)`

Returns the size of the data type *dt* in bytes. 0 is returned for invalid types.

6. `int MLIsValidType(MLDataType dt)`

   Returns `true(=1)` if data type *dt* is valid. Otherwise `false(=0)` is returned.

7. `int MLIsSigned(MLDataType dt)`

   Returns `true(=1)` if data type *dt* is signed. Otherwise `false(=0)` is returned.

8. `int MLIsIntType(MLDataType dt)`

   returns `true(=1)` if data type *dt* is an integer data type. Otherwise `false(=0)` is returned.

9. `int MLIsFloatType(MLDataType dt)`

   Returns `true(=1)` if data type *dt* is a floating point data type. Otherwise `false(=0)` is returned.

10. `int MLIsScalarType(MLDataType dt)`

    Returns `true(=1)` if data type *dt* is a scalar (i.e., a built-in) type. Otherwise `false(=0)` is returned.

There are some more functions for the definition of features and properties related to data types. See the documentation in the file `mlDataTypes.h` for more information. A modern version to get compile-time information on `MLDataType` is to use the `TypeTraits` template class. See the documentation in the file `mlTypeTraits.h` for more information.

- `int MLRangeOrder (MLDataType dt)`
- `int MLHolds (MLDataType dt1, MLDataType dt2)`
- `MLDataType MLGetPromotedType (MLDataType d1, MLDataType d2)`
- `MLDataType MLGetDataTypeForRange (double *min, double *max, int preferUnsigned)`
- `MLDataType MLGetDataTypeForUncorrectedRange (double min, double max, int preferUnsigned)`
- `MLDataType MLGetRangeAndPrecisionEquivalent (MLDataType dt)`
- `MLDataType MLGetPromotedPrecision (MLDataType dt1, MLDataType dt2)`

2.6.4. `MLBase`

This project contains a set of classes that are useful when data structures like markers, lists, functions, diagram information, etc. are needed that are related to image processing, although they may not be an integral part of it. (See also project `MLBase` in the modules library.)

2.6.5. `MLKernel`

A small template class library with some modules for managing a matrix of kernel elements, and for filtering or correlating/convoluting images. See Kernel Progamming for detailed information.

2.6.6. `MLTools`

Class that contains a set of helper functions for different tasks. See `mlTools.h` in project `MLTools` for more information.

2.6.7. MLDiagnosis

The ML project MLDiagnosis contains some modules that can prove to be helpful for module debugging and for changing the ML configuration at runtime.

1. BadModule

   This module is designed to commit a large number of errors and to have many bugs. Hence applications, module networks, MeVisLab, etc. can be checked for stability on bad module behavior.

2. CacheView

   This module shows the current state and load of the Memory Manager cache.

3. Checksum

   This module calculates a checksum of an ML image at the input. This is a simple way to see whether two images differ. Saving just the checksum is sufficient for a later comparison of images. This is especially useful to see whether a module calculates the same image as some time before without storing the entire image.

4. Console

   A module that shows all ML outputs in a console window.

5. CoreControl

   Provides an interface to configure the ML error handling system (e.g., how to handle a fatal error, whether the ML continues or terminates), to enable/disable debugging symbols, to configure the ML caching and multithreading, and to obtain the version of the ML.

6. ErrorTest

   Provides a simple interface to create messages, errors, and exceptions of user defined types. The exact behavior of the applications, error handlers, networks and the ML can be explicitly tested for any type of error.

7. FieldTracer

   In module networks, fields of different modules are often connected and it can become quite difficult to see from where field changes are sent. The field tracer allows for the creation of a field change list in a certain period of time and by that, it allows to analyze changes in the network.

8. MLLogFile

   A module that redirects ML output to a log file. The log file's content can be used for further diagnostic purposes (e.g. after crashes).

9. ModuleView

   This module shows the currently instantiated modules and offers a view on the module interfaces, their fields, inputs and outputs, even when working with an ML release version that does not contain debug information.

10. RuntimeDump

    This module allows for an installation of a dump function in the ML core that will be called when a runtime type causes a crash that is to be handled by the ML. The current state of the C++ interface of some runtime types like fields, modules derived from Module etc. will be dumped in the error

output for further diagnostic purposes. This is intended especially for error diagnostics in release mode when the debugger cannot be used. This module can also remain in released applications so that log information on crashes that did not occur during application development in debug mode are available.

11. `RuntimeView`

    This module shows all currently registered types in the runtime type system as well as the libraries they come from, their parent classes, whether they are abstract or not, etc.

12. `Tester`

    This powerful module applies a number of tests to one or more modules. It checks for correct field names, memory leaks, stable behavior on many different input images with different (page) extents, data types, min/max values, etc. Parameter and base fields are tested with a large number of combinations of values. More test images are continually added to the module. Testing time, intensity, etc. can be controlled by parameters.

13. `TestInput`

    This module generates test images of different types that can be addressed by indices. Thus a large number of different images that cover most image properties like image extent, page extent, min/max values, data type, etc. is available for testing.

2.6.8. `MLImageFormat`

The ML project `MLImageFormat` contains file format classes for storing, loading and modifying an ML `PagedImage` or subimages in a file.

It stores all information of an up to 6D ML `PagedImage`, including extended voxel types, paging information and property extensions. It supports files of more than 4 GB, uses the registered `MLDataCompressors` classes for page-based compression, checks for pages containing only one voxel value to avoid file accesses and unnecessary compressor calls, and many more features. See Section 2.6.9, “ `MLDataCompressors` ” for more information on the `MLDataCompressors`.

The following classes are available as a programming interface (see `mlImageFormatDoc.h` and class headers of those classes for details):

1. `MLImageFormat`

   Class to manage a stored file for saving, loading or retrieving image information. It is mainly used by the module classes.

2. `MLImageFormatTools`

   Collection of independent static file IO classes that are mainly used by `MLImageFormat`.

3. `MLImageFormatFileCache`

   Module class to cache an image in a file comparable to a `MemCache` module.

4. `MLImageFormatSave`

   Module class to save a `PagedImage` and user tags.

5. `MLImageFormatLoad`

   Module class to load a file and some of its information.

6. `MLImageFormatInfo`

Module class to get information about a file.

7. `MLImageFormatTag`

   Tag class used in `MLImageFormatTagList` to store one pair of information items such as a name and an integer or a string.

8. `MLImageFormatTagList`

   Class to describe the list of tag information stored in a file.

2.6.9. `MLDataCompressors`

The ML project `MLDatCompressors` contains the base class `DataCompressor` that allows the implementation of new data compression algorithms. It also contains a factory class `DataCompressorFactory` that allows for the registration of user-derived classes. By that, any number of new compression classes can be implemented which are automatically detected by classes using compression algorithms, for example `MLImageFormat` modules (Section 2.6.8, “ `MLImageFormat` ”).

1. `DataCompressor`

   Abstract base class for ML data compression algorithms. New data compressors can be derived from this class and then be registered in the `DataCompressorFactory` to become available for all other modules and classes that use data compression.

2. `DataCompressorFactory`

   Factory class for ML data compression algorithms. It provides access to all registered data compressors, for example for file formats or memory managers using data compression.

See `MLDataCompressorDoc.h` and other header files in project `MLDataCompressor` for details.

2.6.9.1. How to Implement a New DataCompressor

Follow these steps to implement your own compressor (see `MLDataCompressorDoc.h` and other header files in project `MLDataCompressor` for details and code fragments):

1. Be sure to implement everything in the namespace ML_UTILS_NAMESPACE.

2. Derive your compressor from the `DataCompressor` class and override the following methods:

   a. `virtual const std::string getTypeName() const = 0;`

   b. `virtual const std::string getVersion() const = 0;`

   c. `virtual const bool isSupportedVersion(const std::string &ver) const = 0;`

   d. `virtual MLErrorCode compress(const void *srcMem, size_t srcSize, void *&dstMem, MLint &dstNum) const = 0;`

   e. `virtual MLErrorCode decompress(const void *srcMem, size_t srcSize, void *&dstMem, MLint64 &resSize) const = 0;`

3. Register your `DataCompressor` (for example during dll/so registration) with `YourDataCompressor::initClass()` in the runtime type system of the ML first and then in the `DataCompressorFactory`.

4. Be sure that classes that use your data compressor will find it registered in the `DataCompressorFactory` **before** they are instantiated.

In MeVisLab, you can do this by specifying the PreloadDll flag in a .def file for your compressor.

5. Optionally, you may also want to override the `numUsedHints()` method and initialize the following members appropriately to specify parameters for your compressor which might be detected and passed by some applications to control compression behavior: The parameters *`_hintType`*, *`_hintName`*, *`_rangeMin`*, and *`_rangeMax`* should be set by your derived class, the other parameters should be set to their default values.

6. It is strongly recommended to implement the virtual methods `getVersion()`, `isSupportedVersions()`, `getVendor()`, `getSuffix()`, and `isLossy()` in order to provide additional information about classes using the compressor.

   Especially `isLossy()` should be implemented to make sure that other classes know that decompressed data need not be identical with compressed data. Otherwise, checksum tests done in those classes will fail.

All classes using `DataCompressors` via the `DataCompressorFactory` (the `MLImageFormat` class, for example) will automatically detect your compression algorithm and offer it as an option.

The following example shows a complete header file implementation of a data compressor that packs 16 bit words by removing bit 12 to 15. It is a potentially lossy compressor, because highest bits are removed. It, however, could be useful for CT data, for example, which do not use those bits, or for cases where other compressors do not reach high compression ratios, because data is too noisy:

Example 2.9. CT Data Compressor Packing 12 of 16 Bits

```
#ifndef __mlCTPackDataCompressor_H
#define __mlCTPackDataCompressor_H

#include "MLCTPackDataCompressorSystem.h"
#include "mlDataCompressor.h"

ML_UTILS_START_NAMESPACE

//! CTPackDataCompressor example for the ML.
class MLCTPackDATA_COMPRESSOR_EXPORT CTPackDataCompressor : public DataCompressor
{

public:

  //! Constructor (no destructor needed).
  CTPackDataCompressor() : DataCompressor() { }

  //! Returns name of compression scheme, used e.g., "RLE", or "LZW".
  virtual std::string     getTypeName()     const { return "CTPack";    }

  //! Returns the version string, e.g., "1.1.4" or "1.1"; compatibility
  //! check needs to be done in isSupportedVersion().
  virtual std::string     getVersion()      const { return "1.0";       }

  //! Returns true if the passed version ver is supported by the
  //! implemented compressor class and false otherwise.
  virtual bool            isSupportedVersion(const std::string &ver) const
  { return ver == getVersion(); }

  //! Return the name of the vendor providing the compressor code
  //! or algorithm, something like "MeVis", the author or the
  //! company selling the algorithm.
  virtual std::string     getVendor()       const { return "ML Guide";  }

  //! Returns the suffix describing the compression scheme, for
  //! example "rle" or "lzw".
  virtual std::string     getSuffix()       const { return "cpk";       }

  //! Returns true if compression is lossy, false if not, base class
  //! default is false.
  //! We have to enable lossy, because we throw away highest nibble
  //! which could cause check sum errors in file formats if not denoted.
  virtual bool            isLossy()         const { return true;        }

  //! Number of hints used by the derived compressor class (defaults
```

```
  //! to 0 in base class).
  virtual MLuint8         numUsedHints()    const { return 0;           }
```

```
  //! Compresses a chunk of memory to be decompressed later with decompress().
  //! \param  srcMem  is the pointer of data to be compressed.
  //! \param  srcSize is the size of the data pointed to by srcMem in bytes.
  //! \param  dstMem  the pointer to the compressed data.
  //!                 The compressor will allocate the required memory and
  //!                 overwrites the dstMem pointer which then must be freed
  //!                 by the caller with MLFree() or Memory::freeMemory().
  //! \param  dstNum  returns size of compressed data chunk in bytes or 0 on error.
  //! \return         ML_RESULT_OK on successful compression or an error code
  //!                 describing the error.
  virtual MLErrorCode compress(const  void  *srcMem,
                               size_t        srcSize,
                               void        *&dstMem,
                               MLint        &dstNum) const
  {
    MLErrorCode errCode = ML_NO_MEMORY;
    dstMem = NULL;
    dstNum = 0;

    if (srcMem && (srcSize>0)){

      // Determine size of destination buffer, it requires 4 byte at begin to
      // store original data size and in worst case 1 bit more per CTPack more
      // if no CTPack can be compressed. Add four bytes for rounding securely.

      const size_t packedSize =
        static_cast<size_t>(sizeof(MLint64) + (srcSize * 3) / 4 + 4);

      dstMem = Memory::allocateMemory(packedSize, ML_RETURN_NULL);

      if (dstMem != NULL) {

        // Get byte pointer to output memory and clear data.
        unsigned char *targetData = static_cast<unsigned char *>(dstMem);
        memset(targetData, 0, packedSize);

        // Store size of source data in little endian format at buffer start.
        (static_cast<MLint64*>(dstMem))[0] = static_cast<MLint64>(srcSize);

        if (!MLIsLittleEndian()) {
          MLSwapBytes(targetData, sizeof(MLint64), sizeof(MLint64));
        }

        // Traverse all nibbles/half bytes.
        srcSize *= 2;
        size_t oNibble = 16; // Set start to first nibble after stored startDstSize.
        for (size_t n = 0; n < srcSize; ++n) {

          // Get nibble from source data.
          const unsigned char nib =
            (static_cast<const unsigned char*>(srcMem)[n>>1] >>
             ((n & 1)*4)) & 0xf;

          // Add nibble to output data.
          if ((n & 3) != 3) {
            targetData[oNibble>>1] |= (oNibble & 1 ? (nib << 4) : nib);
            ++oNibble;
          }
        }

        // Calculate number of really used bytes in destination buffer and add 1
        // byte as buffer zone for check of buffer overrun during decompression.
        dstNum = (oNibble >> 1) + (oNibble & 1 ? 1 : 0);

        // Return success.
        errCode = ML_RESULT_OK;
      }
    }
    return errCode;
  }
```

```
  //! Decompresses a chunk of memory created with compress().
  //! \param  srcMem  is the pointer to the compressed data to be decompressed.
  //! \param  srcSize is the size of the data pointed to by srcMem in bytes.
  //! \param  dstMem  returns the pointer to the decompressed data; it is
  //!                 overwritten with the pointer to the allocated and
  //!                 uncompressed data which must be freed by the caller
  //!                 with MLFree() or Memory::freeMemory().
  //! \param  resSize returns the size of the decompressed data
  //!                 memory in bytes or -1 on error.
```

```
  //! \return         ML_RESULT_OK on successful decompression or
  //!                 an error code describing the error.
  virtual MLErrorCode decompress(const  void  *srcMem,
                                 size_t        srcSize,
                                 void        *&dstMem,
                                 MLint64      &resSize) const
  {
    // Pointer to working and result buffers.
    dstMem              = NULL;
    resSize             = -1;
    MLErrorCode errCode = ML_BAD_POINTER_OR_0;

    // Check uncompressed size for at least the four size bytes at start.
    if (srcSize <= sizeof(MLint64)) {
      errCode = ML_FILE_OR_DATA_STRUCTURE_CORRUPTED;
    } else {

      // Get size of decompression data from start of compressed data.
      MLint64 uncompressedSize = (static_cast<const MLint64*>(srcMem))[0];

      if (!MLIsLittleEndian()) {

        // Swap data to local endian format, the data is always stored
        // in little endian.
        MLSwapBytes(reinterpret_cast<unsigned char*>(&uncompressedSize),
                    sizeof(MLint64),
                    sizeof(MLint64));
      }

      if (uncompressedSize < 0) {

        // Should not happen, data is probably corrupted.
        errCode = ML_FILE_OR_DATA_STRUCTURE_CORRUPTED;
      } else {

       // Size seems to be valid, allocate return buffer.
        dstMem = Memory::allocateMemory(static_cast<size_t>(uncompressedSize),
                                        ML_RETURN_NULL);
        if (!dstMem) {
          errCode = ML_NO_MEMORY;
        } else {

          // Unpack all packed nibbles from source data into cleaned result buffer.
          memset(dstMem, 0, uncompressedSize);
          MLint64 oNibble = 0;
          for (size_t n = 16; n < srcSize*2; ++n) {
            // Get nibble from packed data.
            const unsigned char nib =
              (static_cast<const unsigned char*>(srcMem)[n>>1] >>
               ((n & 1)*4)) & 0xf;

            // After unpacking 3 nibbles add a fourth empty one.
            if ((oNibble & 3) == 3) { ++oNibble; }

            // Add nibble to output, shifted by 4 bits if necessary.
            static_cast<unsigned char*>(dstMem)[oNibble >> 1] |=
             (oNibble & 1) ? (nib << 4) : nib;

            ++oNibble;
          }

          // Return success and number of uncompressed bytes.
          errCode = ML_RESULT_OK;
          resSize = uncompressedSize;
        } // else ML_NO_MEMORY;
      } // else if ((uncompressedSize < 0))
    } // else if (srcSize <= sizeof(MLint64) + 1)

    // Clean up on error.
    if (ML_RESULT_OK != errCode) {
      MLFree(dstMem);
      dstMem  = NULL;
      resSize = -1;
    }
    return errCode;
  }


private:

  //! Implements interface for the runtime type system of the ML.
  ML_CLASS_HEADER(CTPackDataCompressor)
};
```

```
ML_UTILS_END_NAMESPACE
#endif // __mlCTPackDataCompressor_H
```

Do not forget to implement the registration code in the ML runtime type system with the typical `ML_CLASS_SOURCE` macro in the .cpp file:

```
ML_CLASS_SOURCE(CTPackDataCompressor, DataCompressor);
```

Also, the registration of the classes in the runtime type system and in the factory for ML data compressors need to be called before using them for the first time (normally in the initialization code while loading the module code):

```
CTPackDataCompressor ::initClass();
DataCompressorFactory::registerCompressor(CTPackDataCompressor::getClassTypeId());
```

2.7. Registered Data Types

Some ML classes are only dedicated to the registration of new voxel data types. They are not part of the ML, but they are registered at initialization time:

1. `MLTypeInfos` - See MLTypeInfos.
2. `MLTStdTypeInfos` - See MLTStdTypeInfos.
3. `MLTComplexTypeInfos` - See MLTComplexTypeInfos.
4. `MLTDoubleVectorTypeInfos` - See MLTDoubleVectorTypeInfos [119].
5. `MLTMatrixTypeInfos` - See MLTMatrixTypeInfos.

2.8. ML Data Types

In the ML, there are some important voxel and data types used in different contexts.

2.8.1. Voxel Types and Their Enumerators

The following voxel types and enumerators are available in the ML:

1. `MLint8` and `MLint8Type`
2. `MLuint8` and `MLuint8Type`
3. `MLint16` and `MLint16Type`
4. `MLuint16` and `MLuint16Type`
5. `MLint32` and `MLint32Type`
6. `MLuint32` and `MLuint32Type`
7. `MLint64` and `MLint64Type`
8. `MLfloat` and `MLfloatType`
9. `MLdouble` and `MLdoubleType`
10. `std::complex<float>` and `MLComplexfType`
11. `std::complex<double>` and `MLComplexdType`
12. `ml::Vector*f` and `MLVector*fType`

13. `ml::Vector*d` and `MLVector*dType`
14. `ml::Matrix*f` and `MLMatrix*fType`
15. `ml::Matrix*d` and `MLMatrix*dType`
16. `ml::Vector*i8` and `MLVector*i8Type`
17. `ml::Vector*i16` and `MLVector*i16Type`
18. `ml::Vector*i32` and `MLVector*i32Type`
19. `ml::Vector*i64` and `MLVector*i64Type`

2.8.2. Index, Size and Offset Types

The following index and offset types are available in the ML:

1. `MLint`

   A signed ML integer type with at least 64 bits used for all index calculations on very large images even on 32 bit systems (typically used for positions and coordinates in images). It is widely used in the ML for 64 bit index, size and range specifications which support signed arithmetic.

   Examples of usage are classes `ImageVector` and `SubImageBox` which use `MLint` as members.

2. `MLuint`

   An unsigned ML integer type with at least 64 bits used for index calculations on very large images even on 32 bit systems. Its is sometimes needed for image positions and coordinates where a sign is not desired. Note that the signed `MLint` should normally be used for safe signed arithmetics, so `MLuint` is rarely used in ML contexts. It is typically used for index, size and range specifications without sign.

   An example of usage is the specification of file sizes which can be larger than 4 GB even on 32 bit systems and where negative sizes make no sense. Further examples are some function arguments in `mlFileSystem.h` .

3. `MLsoffset`

   This signed ML offset type is a 32 bit signed integer on 32 bit platforms and a 64 bit signed integer on 64 bit platforms. This type is typically used for expressions in pointer offsets where 64 bit integers could cause warnings on 32 bit systems (because their range exceeds 32 bit pointer offsets). Such a type is necessary, because normal integers are not large enough on 64 bit systems; they remain 32 bit on most 64 bit platforms. In most ML sources, the `MLsoffset` type is used instead of the `MLuoffset`, because signed arithmetic is often required in image processing operations.

   An example of usage are index tables for kernel elements which are added to pointers (see project `MLKernel` and classes `KernelBaseModule` and `KernelModule`). These indexes must be able to describe negative and positive offsets on pointers which remain in the address space of the system.

4. `MLuoffset`

   This unsigned ML offset type is a 32 bit unsigned integer on 32 bit platforms and an unsigned 64 bit one on 64 bit platforms. Such a type is necessary, because a normal unsigned integer is not large enough on 64 bit systems; it remains 32 bit even on many 64 bit platforms. Be careful when using this type in ML image processing, because in most contexts signed arithmetic is required when offsetting image pointers. Thus it is rarely used in ML contexts.

5. `MLssize_t`

The signed ML size type is a signed 32 bit `size_t` on 32 bit platforms and 64 bit `size_t` on 64 bit platforms. It corresponds to the normal `ssize_t` type on Unix platforms and to the `SSIZE_T` type on windows platforms. It is used for index, size and range specifications which do not exceed the signed range of the address space of a system. It is rarely used in ML contexts.

6. `MLsize_t`

   The unsigned ML size type is an unsigned 32 bit `size_t` on 32 bit platforms and 64 bit `size_t` on 64 bit platforms. It corresponds to the normal `size_t` type available on most systems. It is typically used for index, size and range specifications which do not exceed the range of the address space of a system. The original `size_t` type is used in most ML code, because it is platform-independent.

Note

Internally (on 32 bit platforms), a `size_t` (and `MLsize_t`) is normally either an `unsigned int` or an `unsigned long` depending on compilers and platforms.

`MLuoffset` is always an unsigned integer type. Therefore `size_t` and `MLuoffset` do not behave identically everywhere (for example when they are passed as references) although their sizes and signs always come along with each other. So both types are useful and in a few cases they have to be distinguished carefully when implementing platform-independent code.

Of course, the same applies for the types `MLssize_t` and `MLsoffset`.

Chapter 3. Deriving Your Own Module from Module

Chapter Objectives

By reading this chapter you will learn how to derive your own ML module from the class `Module`. You will receive detailed information on the following methods:

- *Constructor*,
- *Destructor*,
- `activateAttachment`,
- `handleNotification`,
- `calculateOutputImageProperties`,
- `calculateInputSubImageBox`,
- using `TypedCalculateOutputImageHandler`,
- `calculateOutputSubImage`,
- `handleInput`,
- `getTile`,
- `getUpdatedInputImage`.

Also, you will learn how to use and configure additional functionality, such as:

- checking for interruptions,
- multi-threading,
- bypassing page data, and
- activating the support of registered voxel types.

With MeVisLab version 2.2, a new concept to separate module functionality from image processing functionality has been introduced in the form of using a `TypedCalculateOutputImageHandler`. Read Section 3.1.5, “Using `TypedCalculateOutputImageHandler`” to learn more.

The chapter ends with a discussion of typical traps and pitfalls you may encounter when you implement classes derived from `Module`. See Section 3.1.18, “Traps and Pitfalls in Classes Derived from Module ”.

See Section A.1, “Creating an ML Project by Using MeVisLab”. for a quick start with module development.

Important

The ML module wizard in MeVisLab supports many of the steps discussed in the following sections. Use the wizard in order to avoid spending too much time on writing everything on your own!

3.1. Deriving from `Module`

The following sections will explain how to implement your own image processing algorithm.

3.1.1. Basics

When you begin to implement your own ML image processing module, you usually just need the following include file:

```
#include "mlModuleIncludes.h"
```

All ML specific C++ code should be written within the namespace ML - thus no prefixes are needed before constants and classes, and collisions with other library symbols are minimized:

```
ML_START_NAMESPACE

  // here the ML specific code is added

ML_END_NAMESPACE
```

An image processing module is derived from the class `ml::Module`. Since modules are usually compiled in their own *DLL* (Windows: *"dynamic linked library*", Linux: *"shared library"*, Mac OS: "*dynamic shared libraries*"), it may be necessary to export this class on the DLL interface. Therefore, a macro `MLEXAMPLEOPSEXPORT` is used to specify the export of a class in the system header file of the DLL. See Section A.3, “Exporting Library Symbols”

```
ML_START_NAMESPACE

  class MLEXAMPLEOPSEXPORT AddExample : public Module
  {
    // class interface and/or code

  }; // end of class AddExample

ML_END_NAMESPACE
```

Note

Although exporting classes is only necessary on Windows platforms, it should be added while developing on other platforms as well in order to ensure platform-independence.

Since a new ML module is usually compiled as a new library that an application can load at runtime, you must make your module accessible to a module database. The ML implements such a database as a *Runtime Type System* (see also Section 2.2.4, “The Runtime Type System”). Thus implementing your own module just requires a small interface to enter the module as a new type in that runtime type system. Hence, it can give its name and its type on request as well as create an instance of itself on demand. The following macro (from file `mlRuntimeSubClass.h`) declares the necessary class interface:

```
ML_START_NAMESPACE

  class MLEXAMPLEOPSEXPORT AddExample : public Module
  {
    // class interface and/or code ...

    // Implement runtime type interface of module. Add it at
    // end of class declaration since it changes member access
    // control to  'private'.

    ML_MODULE_CLASS_HEADER(AddExample)

  }; // end of class AddExample

ML_END_NAMESPACE
```

Important

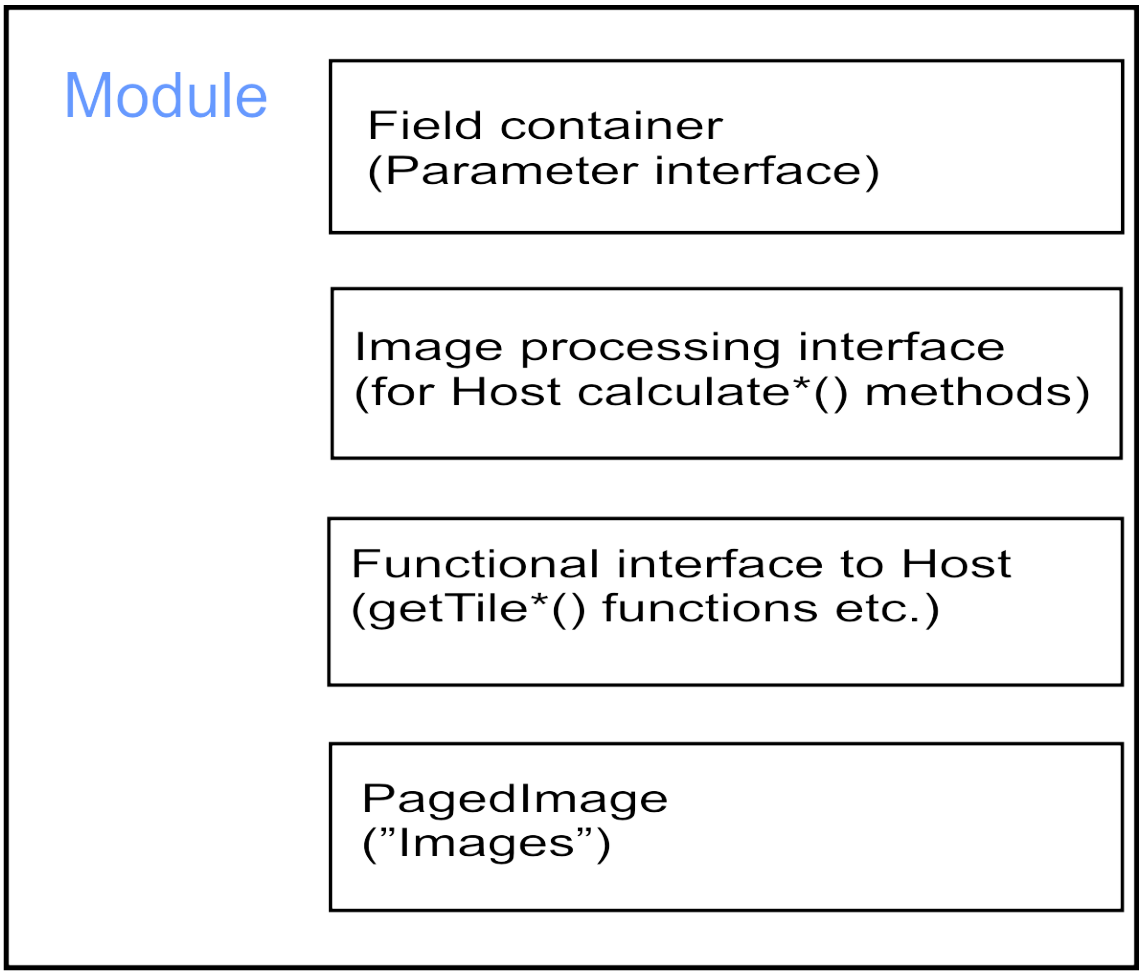
To make this class available to the runtime type system it is necessary to call its static `init()` function. This function will be declared by this macro when the dynamic linked library of your module is initialized.

Important

Be sure that the class name is written correctly, since not all compilers are able to check for wrong names in that macro.

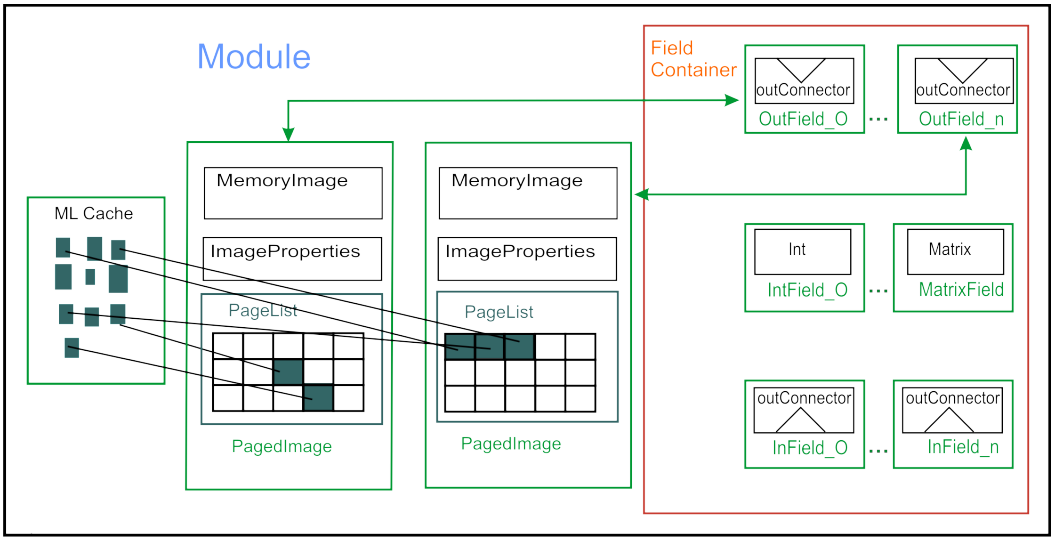
A simple overview of a `Module`:

Figure 3.1. `Module` Structure (I)



The `Module` is derived from `FieldContainer` that holds the module's parameters:

Figure 3.2. `Module` Structure (II)



3.1.2. Implementing the Constructor

The constructor is a crucial part of an ML module because it

- generates the parameter interface (including inputs and outputs) and also initializes it,
- enables/disables multithreading support,
- specifies whether your module performs in-place calculations or bypasses image data,
- can specify how changes to the parameter interface (including inputs) notify output images.

The implementation of the constructor must always include a base class constructor call of the class `Module`, and the number of image inputs and output a module is passed as arguments (two inputs and one output in the example):

```
ML_START_NAMESPACE

  AddExample::AddExample(): Module(2,1)
  {
    // ...
  }
```

```
ML_END_NAMESPACE
```

See also class `FieldContainer` (Section 2.1.3, “ FieldContainer ”) as well as classes `InConnectorField` and `OutConnectorField` (Section 2.1.2, “ Field ”) for other ways of adding or removing inputs to/from your modules.

Now a set of parameters can be added to specify the module interface. Note that all fields are added to the module (see also class `FieldContainer`).

Be aware that field names should only use alphanumeric characters and may not include spaces or special characters. The example code fragment adds a float and a Boolean parameter to the module interface and initializes them:

```
_addConstFld   = addFloat("Constant");
_addConstFld   ->setFloatValue(0);

_deleteVoxelFld= addToggle("DeleteVoxel");
_deleteVoxelFld->setIntValue(false);
```

Programmers who favor short code can also write the following:

```
(_addConstFld   = addFloat("Constant") )->setFloatValue(0);
(_deleteVoxelFld= addToggle("DelVoxel"))->setIntValue(false);
```

Note that the members *`_addConstFld`* and *`_deleteVoxelFld`* are pointers to the field types that are created, and must be declared in the header file like this:

```
private: // or protected

  FloatField  *_addConstFld;
  ToggleField *_deleteVoxelFld;
```

Access functions can be implemented to make fields and module parameters directly accessible to an application without permitting field pointer changes. These functions are especially useful when further classes are to be derived from your class without the risk of derived classes doing modifications to invalid field pointers:

```
public:

  inline FloatField  &getAddConstFld()    const { return *_addConstFld;    }
  inline ToggleField &getDeleteVoxelFld() const { return *_deleteVoxelFld; }
```

If you want parameter changes to also invalidate the image output of the module and to notify connected modules of the changed/invalidated image, you can simply connect your field(s) to the changed output image:

```
_addConstFld   ->attachField(getOutputImageField(0));
_deleteVoxelFld->attachField(getOutputImageField(0));
```

Important

If not disabled, field value changes notify all observers of the field. Therefore the `handleNotification()` function of your module is also called when you set field values.

The following two methods (they may be nested) can be used to avoid the `handleNotification()` being called when field values are set:

```
handleNotificationOff();

  // Change field values here without calling handleNotification().

handleNotificationOn();
```

Note

Input and output images are also ML module parameters and therefore they are represented by fields (`InputConnectorField` and `OutputConnectorField`) as well.

Since input and output fields can be added via the superclass constructor, the methods `getInputImageField(int idx)` and `getOutputImageField(int idx)` are available to access these fields.

Usually, input image changes need to invalidate the output and to notify the connected modules; if so, the output field(s) just have to be attached to the input field(s). This is possible because fields handle input images like other module parameters:

```
getInputImageField(0)->attachField(getOutputImageField(0));
getInputImageField(1)->attachField(getOutputImageField(0));
```

Some additional features of the `Module` class allow the configuration of further image processing behavior:

- In-place image processing. See Section 3.1.11.1, “Inplace Image Processing”.
- Bypassing image data. See Section 3.1.11.2, “Bypassing Image Data”.
- Processing image data in parallel. See Section 3.1.11.3, “Multithreading: Processing Image Data in Parallel”.
- Processing images of registered voxel types. See Section 3.1.11.4, “Processing Images of Registered Voxel Types”.

See Section 3.1.11, “Configuring Image Processing Behavior of the Module” for further details.

3.1.3. Module Persistence and Overloading `activateAttachments()`

In common ML modules, the algorithm's parameters are implemented as fields. Therefore, module persistence does normally not have to be implemented, since the application usually should scan the field interface of all ML modules as well as save and reload their states from/to a file (see also Section 2.1.2, “ `Field` ” and Section 2.1.3, “ `FieldContainer` ”).

When an application reloads or clones ML modules, a specific problem needs particular attention. Within the given situation, the application and its connections usually re-create the network modules, and field values are restored. This causes some network modules to start calculation, because fields are updated by the loading process, which would not only result in long startup times but also in calculations being performed on partially invalid module data.

The solution to this problem is to disable field notifications while loading (`handleNotification()` and other field observers are not called) and to notify all modules with a "load-finish" signal when loading has been completed. So the modules can update their internal states to the new field values in one step. Many modules do not need to handle this signal, but some do. To implement this update functionality, the method `activateAttachments()`that stands for this "load finished" signal can be overloaded:

```
virtual void activateAttachments()
{
  // Implement your update stuff here ...

  // Do not forget to call the super class functionality, it enables field
  // notifications for your module again.
  // SUPER_CLASS is the class you derive from (usually Module).

  SUPER_CLASS::activateAttachments();
}
```

As a general rule, you need to overload this method when your class includes non-field members that require updates on field changes. Update these members in `activateAttachments`, because there you have the new field setting after e.g., module reloads.

Note

The order of execution on loading a module is as follows:

1. Module creation (constructor call)

2. Loading and setting of field values and connections (without calls of `handleNotification(Field*)`)

3. Call of `activateAttachements()`

3.1.4. Implementing `handleNotification()`

Sometimes it is necessary to react on changes to the fields that represent a module's interface. This can easily be done by overloading the method `handleNotification()` which is called when any field (value) is changed.

```
void AddExample::handleNotification(Field *field)
{
  if (field == _addConstFld) {
    /* The value of _addConstFld has changed. */
  }
  if (field == getInputImageField(0)) {
    /* First  input is (dis)connected, updated, invalidated... */
  }
  if (field == getInputImageField(1)) {
    /* Second input is (dis)connected, updated, invalidated... */
  }
}
```

Note

The `handleNotification()` call should be carefully observed, because:

- Any change to field values (also from within the constructor!) normally causes a call of this method (if not blocked). See `handleNotificationOff()` and `handleNotificationOn()` as described in Section 3.1.2, “Implementing the Constructor”. The call of `handleNotification()` is deactivated between these two calls which is useful e.g., in the constructor to avoid side effects during the initialization phase of the module.

- Field changes from inside of `handleNotification()` do not cause recursive `handleNotification()` calls in the same module because that is usually not desired. Nevertheless, such field updates can cause `handleNotification()` calls in other modules (e.g., via field connections).

- Changing fields within `calc*` methods is generally allowed but these methods never call `handleNotification()` and do not notify connected fields. This is necessary to avoid image processing being indirectly restarted by field updates.

- In the overloaded method `handleNotification(Field *f)`, it is not needed to call the superclass code since `Module::handleNotification()` is an empty method.

Tip

The statement

```
if (field==_addConstFld) { getOutputImageField()->touch(); }
```

in `handleNotification()` usually has the same effect as

```
_addConstFld->attachField(getOutputImageField());
```

in the constructor.

3.1.5. Using `TypedCalculateOutputImageHandler`

Since MeVisLab version 2.2, a new way to implement typed image processing in an ML module has been introduced which is the default setting of MeVisLab's module wizard. It uses a separate class for the actual image processing which is derived from `TypedCalculateOutputImageHandler`.

Using a `TypedCalculateOutputImageHandler` has the following advantages:

- It supports complex configurations of output/input type combinations (compared to the CALC_* macros).
- It facilitates implementation of thread-safe image processing, since the processing is no longer done in the module itself.
- It allows to have different output image handlers for different output images or even for different module states.

For further information, please read ml::TypedCalculateOutputImageHandler, ml::CalculateOutputImageHandler, and ml::Module::createCalculateOutputImageHandler.

3.1.6. Implementing `calculateOutputImageProperties()`

The virtual method `calculateOutputImageProperties(int outIndex, PagedImage* outImage)` must be overloaded to change the properties of the output images, as well as to change the properties of the input subimages which are passed to `calculateOutputSubImage()`.

For a certain output index, the method sets properties of the output image (depending on the properties of the input images). Hence, for each property of the output image *`outImage`*, the corresponding properties of any input image `getInputImage(0)`, ... , `getInputImage(getNumInputImages()-1)` can be merged and set as new properties.

To change the properties of an input subimage, you can use the following methods of the `PagedImage`:

- `void setInputSubImageDataType(int inputIndex, MLDataType datatype)`
- `void setInputSubImageIsReadOnly(int inputIndex, bool readOnly)`
- `void setInputSubImageUseMemoryImage(int inputIndex, bool useMemImg)`
- `void setInputSubImageScaleShift(int inputIndex, const ScaleShiftData& scaleShift)`

An access method to the input images is available with `getInputImage(int index)`.

Note

Do not use `getOutputImage(int index)` from within `calculateOutputImageProperties()` and it is not allowed to change the properties of other output images than the one obtained as an argument.

Note

In case of `processAllPages(-1)`, the *`outIndex`* will equal -1 and *`outImage`* will be the temporary PagedImage.

Input images and their properties within the `calculateOutputImageProperties()` and `calculate*()` methods are always valid and thus do not have to be checked for validity.

Access methods to the image properties are defined in the classes `ImageProperties` (Section 2.3.1, “ `ImageProperties` ”), `MedicalImageProperties` (Section 2.3.2, “ `MedicalImageProperties` ”) and `PagedImage` (Section 2.3.4, “ `PagedImage` ”):

- `getImageExtent()` and `setImageExtent()`,
- `getBoxFromImageExtent()`,
- `getPageExtent()` and `setPageExtent()`,

- `getDataType()` and `setDataType()`,
- `getMinVoxelValue()` and `setMinVoxelValue()`,
- `getMaxVoxelValue()` and `setMaxVoxelValue()`,
- and many more.

If `calculateOutputImageProperties()` is not implemented, the properties of `getInputImage(0)` are copied to the output image(s).

The following example shows how to set some of the most important properties of an output image.

```
void ExampleModule::calculateOutputImageProperties(int outIndex, PagedImage* outImage)
{
  // Set image extent
  outImage->setImageExtent ( ImageVector(100,100,30,3,1,1) );

  // Set page extent
  outImage->setPageExtent( ImageVector(128,128,1,1,1,1) );

  // Set estimated min voxel value
  outImage->setMinVoxelValue(   0 );

  // Set estimated max voxel value
  outImage->setMaxVoxelValue( 255 );

  // Set desired data type
  outImage->setDataType(MLuint8Type);
}
```

Note

Setting minimum and maximum voxel values can sometimes be a difficult task because page-based algorithms usually do not process the entire image and explicit testing of all voxel values is impossible. Therefore the typical approach to solve this problem is to set minimum and maximum voxel values in such a way that they include all voxel values that *could* occur. The minimum/maximum range can be set to be larger than the real voxel values in order to make things easier even when the minimum and maximum values become very large. These values are considered to be hints and no reliable values. However, the maximum value must always be equal to or greater than the minimum value.

When setting the properties of the output image(s), the following should be considered:

- Changing properties of output images is only legal inside the `calculateOutputImageProperties()` method.
- Page extends must be left unchanged unless it is really necessary to avoid performance drawbacks. They must not set to the image's extend, since pages are usually inherited by subsequent modules, and setting a too large page extend will degenerate the underlying page concept.
- The following code fragment must be used to invalidate/validate the output image at index `outIdx`:

  ```
  // Invalidate the output image.

  outImage->setInvalid();


  // Validate the output image.

  outImage->setValid();
  ```

  This is only to be used in `calculateOutputImageProperties()`

3.1.7. Implementing `calculateInputSubImageBox()`

Before the algorithm can calculate the contents of an output page, the required data portion / block from each input must be specified in `calculateInputSubImageBox()`. The algorithm must return

that subimage region of the image at input *inIndex* that is needed to calculate the subimage region *outSubImgBox* of the output at index *outIndex*:

```
virtual SubImageBox calculateInputSubImageBox(int inIndex,
                                    const SubImageBox& outSubImgBox,
                                    int outIndex)
{
  // Do the same for all inputs and outputs:
  // Get corners of output subimage.

  const ImageVector v1 = outSubImgBox.v1;
  const ImageVector v2 = outSubImgBox.v2;


  // Request a box from input image which is shifted by 10 voxels to the left
  // and 5 voxels to the front.

  return SubImageBox(ImageVector(v1.x-10, v1.y-5, v1.z, v1.c, v1.t, v1.u),
                     ImageVector(v2.x-10, v2.y-5, v2.z, v2.c, v2.t, v2.u));
}
```

The code is shorter when vector arithmetics are used:

```
virtual SubImageBox calculateInputSubImageBox(int inIndex,
                                    const SubImageBox& outSubImgBox,
                                    int outIndex)
{
  // Request a box from input image which is shifted by 10 voxels to the
  // left and 5 voxels to the front.

  return SubImageBox(outSubImgBox.v1+ImageVector(-10, -5, 0,0,0,0),
                     outSubImgBox.v2+ImageVector(-10, -5, 0,0,0,0));
}
```

If `calculateInputSubImageBox` is not implemented, the default implementation returns the unchanged *outSubImgBox*, i.e., if a certain region of the output image is calculated, the same region is requested from the input image.

Note

Requesting areas outside the input image is explicitly legal because this is often useful when input regions need to be bigger than output regions, e.g., for kernel-based image processing (Section 4.2.4, “Kernel-Based Concept”). However, image data requested from outside an image region will be undefined.

3.1.8. Changes to `calcInSubImageProps()`

As with MeVisLab version 2.1, this method has been removed.

The properties of the input subimages (typically changes to the data type of in the input data before processing them) need to be set now in the method `calculateOutputImageProperties()`. This way, the properties of the input subimages are set only once for each output image and not for each input subimage request. Thus, the new way is faster and less error prone.

3.1.9. Implementing `calculateOutputSubImage()`

The ML calls this method to request the calculation of real image data or, to be more precise, to request the calculation of one output page.

In *outSubImg*, a pointer to a page of output image *outIndex* is passed. The contents of that page need to be calculated by the algorithm.

In *inSubImgs*, the pointers to the input subimages are passed. These subimages contain the source data and exactly the same image regions you requested in `calculateInputSubImageBox()` for the output of index *outIndex*. Note that the number of input subimages depends on the number of module inputs; this number can be 0 if there are no module inputs (e.g., a `ConstImg` or a `Load` module).

```
virtual void calculateOutputSubImage(SubImage *outSubImg, int outIndex, SubImage *inSubImgs){ ... }
```

The data types of the input and output data can be any of the types supported by the ML, i.e., 8,16,32 or 64 bit integers, float, double or any of the registered data types. Implementing the algorithm to support all these data types is generally difficult, especially because it is not known whether future ML versions will contain other data types.

The solution to this problem is to implement a template function that is automatically compiled for all data types. This, however, requires the correctly typed template function to be called from `calculateOutputSubImage()`. This should not be implemented by the module developer because additional data types and optimizations could change that process.

A set of predefined macros is available, e.g., the following can be used if there is one module input and the template function must be implemented in the C++ file.

```
ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_SCALAR_TYPES_CPP(ExampleModule);
```

The correct macro is built from

- the string `ML_CALCULATE_OUTPUTSUBIMAGE`,
- the number of image inputs right behind this, coded as the string `_NUM_INPUTS_*`, where `*` is one of `0`, `1`, `2`, `3`, `4`, `5`, `10` or `N`,
- the string `_SCALAR_TYPES` or `_DEFAULT_TYPES` if the data types of input and output subimages are the same and the module shall either only support scalar types or the default voxel type set, or
- the string `_DIFFERENT_SCALAR_INOUT_DATATYPES` or `_DIFFERENT_DEFAULT_INOUT_DATATYPES` if different data types of input and output subimages shall be allowed (requires using `PagedImage::setDataType()` and `PagedImage::setInputSubImageDataType()` in the `calculateOutputImageProperties()` method). Again this either only supports only scalar types or the default voxel type set. Note that all the subimages for each input image still must have the same data type, only the types between input and output subimages can differ,
- alternatively you can have the string `_WITH_CUSTOM_SWITCH` (or `_DIFFERENT_INOUT_DATATYPES_WITH_CUSTOM_SWITCH`) if a subset of certain data types shall be allowed only as input data types. There are a number of predefined macros for the switches available, such as `ML_IMPLEMENT_FLOAT_CASE` for all floating point data types or `ML_IMPLEMENT_COMPLEX_CASES` for complex data types, and the user can implement new data type switches as well.
- The whole macro must end with the string `_CPP` if the C++ file implementation is used. If the header file implementation is used, no special ending string needs to be provided.
- As arguments for the macro, the class name of the module needs to be provided and if the macro should support a subset of custom data types, the macro that implements the switch for those data types needs to be provided as well.

Note

- Particular attention must be paid to the exact name of the template function implemented for the macro (`calculateOutputSubImage` or `calculateOutputSubImage` **`T`**), as well as to its number of template and parameter arguments to avoid annoying compilation problems.

  Many compilers only check the signature of the template functions in the header file; so **it must be made sure that the function signatures in the header and cpp files are identical**.

- One might expect macros for more than two data types: type 1 for the output subimage, type 2 for input subimage 0, and type 3 for input subimage 3. If you need this degree of control, you should switch to typed output handlers, which are more flexible in this regard.

It is also possible to specify a subset of data types (e.g., only integer, only float data, only standard data types) which will not be discussed here. See Section 7.5.3, “Reducing Generated Code and Compile Times” and the file `mlModuleMacros.h` for more information.

Important

If not specified otherwise, the input subimages have always the same data type as the output subimages. However, the data type for the input subimages can the changed for each input image.

To change the data type for a certain input image (and therefor for each of its subimages), you need to implement this in the method `calculateOutputImageProperties()`.

The template function with the algorithm can be implemented as follows:

```
template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                    int outIndex,
                                    TSubImage<DATATYPE> *inSubImg0,
                                    TSubImage<DATATYPE> *inSubImg1)
{
  //...
}
```

In this template function, the algorithm calculates the output page *`outSubImg`* from the input page(s) *`inImg1`* and *`inImg2`*. This method is instantiated for each data type. The method `calculateOutputSubImage` calls this function by searching the correct data type and by calling the correctly typed template version. It is automatically implemented by the corresponding `ML_CALCULATE_OUTPUTSUBIMAGE` macro.

The number of typed input subimages depends on the used macro, e.g., for zero inputs

```
template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg, int outIndex)
{
  //...
}
```

for four inputs

```
template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                    int outIndex,
                                    TSubImage<DATATYPE> *inSubImg0,
                                    TSubImage<DATATYPE> *inSubImg1,
                                    TSubImage<DATATYPE> *inSubImg2,
                                    TSubImage<DATATYPE> *inSubImg3)
{
  //...
}
```

and for a dynamic number of inputs

```
template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                        int outIndex,
                                        TSubImage<DATATYPE> **inSubImgs)
{
  //...
}
```

For two inputs and different input and output image data types:

```
template <typename ODTYPE, typename IDTYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<ODTYPE> *outSubImg,
                                    int outIndex,
                                    TSubImage<IDTYPE> *inSubImg0,
                                    TSubImage<IDTYPE> *inSubImg1)
{
  //...
}
```

This copies voxel by voxel from the input subimage to the available output subimage, e.g., with the macro `ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_SCALAR_TYPES_CPP(SubImgExampleModule)`:

```
template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                    int /*outIndex*/,
                                    TSubImage<DATATYPE> *inSubImg0)
{
  // Copy overlapping data from inSubImg0 to outSubImg.
  outSubImg->copySubImage(*inSubImg0);
}
```

Note that the classes `TSubImage` and its base class `SubImage` provide a number of other typed and untyped copy, fill and access methods for subimages and their data.

This implements a voxel-wise copy from the input subimage to the output image, keeping track of the coordinate of the copied voxel:

```
template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                    int /*outIndex*/,
                                    TSubImage<DATATYPE> *inSubImg0)
{
  // Determine overlapping and valid regions of page and image, because the
  // page could reach outside valid image region.

  const SubImageBox box = inSubImg0->getValidRegion();

  // Traverse all voxels in box
  ImageVector p = box.v1;
  for (p.u = box.v1.u;  p.u <= box.v2.u; ++p.u) {
    for (p.t = box.v1.t;  p.t <= box.v2.t; ++p.t) {
      for (p.c = box.v1.c;  p.c <= box.v2.c; ++p.c) {
        for (p.z = box.v1.z;  p.z <= box.v2.z; ++p.z) {
          for (p.y = box.v1.y;  p.y <= box.v2.y; ++p.y) {

            // Set x coordinate of first voxel in row.

            p.x = box.v1.x;


            // Get pointer to input voxel at position p.

            const DATATYPE * inPtr0 = inSubImg0->getImagePointer(p);
            DATATYPE *       outPtr = outSubImg->getImagePointer(p);


            // Implement inner loop without function calls and use
            // pointer iterations for a better performance.

            for (;  p.x <= box.v2.x;  ++p.x) {
              *outPtr = *inPtr0;  // Copy input voxel to output voxel.
              ++outPtr;           // Move both voxel pointers forward.
              ++inPtr0;
            }
          }
        }
      }
    }
  }
}
```

The following code fragment shows the implementation for one input and one output of different types for input and output subimages. The macro

```
ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_DIFFERENT_SCALAR_INOUT_DATATYPES_CPP(ExampleModule)
```

is used for that in addition of change of the data type in the `calculateOutputImageProperties` method:

```
//! Select either MLint64 or MLdouble as output type.
void ExampleModule::calculateOutputImageProperties(int outIndex, PagedImage* outImage)
{
  if (MLIsIntType(outImage->getDataType()))
  {
    // Use int64 instead of any other int type.
    outImage->setDataType(MLint64Type);
  }
  else
  {
    // Use double for all other types.
    outImage->setDataType(MLdoubleType);
  }
```

```
  // Set the data type of the input image to the input subimages
  // instead of the data type of the output image (which is the default).
  outImage->setInputSubImageDataType(0, getInputImage(0)->getDataType());
}


//! Implement the calls of the right template calculateOutputSubImage code
//! for the current image data type for all data type combinations.

ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_DIFFERENT_SCALAR_INOUT_DATATYPES_CPP(ExampleModule);


template <typename ODTYPE, typename IDTYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<ODTYPE> *outSubImg,
                                    int /*outIndex*/,
                                    TSubImage<IDTYPE> *inSubImg0)
{
  // Determine overlapping and valid regions of page and image, because the
  // page could reach outside the valid image region.

  const SubImageBox box = inSubImg0->getValidRegion();

  // Traverse all voxels in the box

  ImageVector p = box.v1;
  for (p.u = box.v1.u;  p.u <= box.v2.u;  ++p.u) {
    for (p.t = box.v1.t;  p.t <= box.v2.t;  ++p.t) {
      for (p.c = box.v1.c;  p.c <= box.v2.c;  ++p.c) {
        for (p.z = box.v1.z;  p.z <= box.v2.z;  ++p.z) {
          for (p.y = box.v1.y;  p.y <= box.v2.y;  ++p.y) {

            // Set x coordinate of first voxel in row and
            // get pointer to input voxel at position p.

            p.x = box.v1.x;
            const IDTYPE * inPtr0 = inSubImg0->getImagePointer(p);
            ODTYPE *       outPtr = outSubImg->getImagePointer(p);


            // Implement inner loop without function calls and use
            // pointer iterations for a better performance. Use a
            // cast to convert voxel types warn free.

            for (;  p.x <= box.v2.x;  ++p.x)
            {
              // Copy input voxel to output voxel.

              *outPtr = static_cast<ODTYPE>(*inPtr0);

              // Move both voxel pointers forward.

              ++outPtr;
              ++inPtr0;
            }
          }
        }
      }
    }
  }
}
```

An implementation with fixed input/output types and without templates or macros is also possible if the programmer takes care of correct `TSubImage` for the fixed types:

```
//! Always select MLint32 as output type.
void ExampleModule::calculateOutputImageProperties(int outIndex, PagedImage* outImage)
{
  outImage->setDataType(MLint32Type);

  // Always select MLdouble as voxel type for input subimages.
  outImage->setInputSubImageDataType(0, MLdoubleType);
}


//! Implement explicitly the copy from the double typed input
//! buffers to the int32 typed output subimage.
void ExampleModule::calculateOutputSubImage(SubImage *outSubImg,
                                    int /*outIndex*/,
                                    SubImage *inSubImgs)
{
  // You can use either the untyped copySubImage() method:
```

```
  outSubImg->copySubImage(inSubImgs[0]);


  // ... or build typed subimages from the untyped ones and implement
  // loops as in previous examples on typed oSubImg and iSubImg.

  TSubImage<MLint32>  oSubImg(*outSubImg);
  TSubImage<MLdouble> iSubImg(inSubImgs[0]);

  //  ...  implement voxel loop as in previous examples here
}
```

See Section 3.1.17, “Processing Input Images Sequentially” and Section 7.2.3, “Examples with Registered Voxel Types”, and ML example codes in MeVisLab for further and advanced examples of `calculateOutputSubImage()` implementations.

Important

Subimages contain a set of image properties that can be useful for programming. However, it would require a significant effort to calculate the `minVoxelValue()`, `maxVoxelValue()` and `isValid()` properties correctly for each `calculateOutputSubImage()` call, and therefore they are neglected. They must be retrieved from input image `getInputImage(inImgIdx)` when needed.

Tip

Have a close look at the class `TSubImage` (Section 2.3.5, “`SubImage/TSubImage`”) and at Chapter 4, *Image Processing Concepts* before you begin to implement more functionality in `calculateOutputSubImage()`. Most of the standard functionality, like subimage and voxel filling, copying, overlapping, cursor positioning, value reading/setting, etc. are already implemented there and can be used to simplify your work considerably.

Many problems (and solutions) like global input image access in pages etc. are discussed there as well.

3.1.10. Handling Disconnected or Invalid Inputs by Overloading `handleInput()`

By default, a module's `Module::calculate*` methods are not called when any of its input images are disconnected or connected to an invalid image. This, however, is desired in some cases, e.g., to support optional input images or when implementing a `Switch` module that has multiple inputs and only a few of them are connected and valid, while only one of the images shall be passed to the output image.

To support disconnected or invalid input images, one has to overload the following method:

```
virtual INPUT_HANDLE handleInput(int inIndex, INPUT_STATE state) const;
```

Whenever an input is disconnected or invalid while it is being accessed, the ML internally calls `handleInput()` with the current input state and requests how to handle this situation. ask for a task with that input. There are some cases to be handled when input at index *`inIndex`* is accessed via a `Module` method:

- The input is connected and valid.

  Normal image processing takes place, and the `handleInput()` method is not called.

- The input is disconnected or connected but invalid after trying to update its properties. This case is notified by the parameter *`state`* with value `DISCONNECTED` or `CONNECTED_BUT_INVALID`.

  There are two possibilities:

  - `handleInput()` returns `INVALIDATE` and no image processing can take place (which is the default).

- `handleInput()` explicitly allows an invalid input image by returning `ALLOW_INVALID_INPUT`. Image processing will continue and `getInputImage(inIndex)` will return NULL for that index. The ML Host will not request data from this image and subimages passed to `calculateOutputSubImage` will be empty for that input image.

Important

`handleInput()` must return a unique value for each input configuration. Input handling cannot change during the lifetime of the `Module` instance. If it changes, image processing may become instable.

Note

Disconnected and connected but invalid inputs can be handled differently by using the passed `state`, although is does not make sense in most situations.

3.1.10.1. Checking Module Inputs for Validity

When looking at a module's input, it may have one of the following states (of enum type `INPUT_STATE`):

- `DISCONNECTED` - no image is connected.
- `CONNECTED_AND_VALID` - an image is connected and it is valid.
- `CONNECTED_BUT_INVALID` - an image is connected but invalid, even after trying to update its properties.
- `CONNECTED_BUT_NEEDS_UPDATE` - an image is connected but its properties are out of date and need updating. After the update, it may become valid or invalid.

This state can be requested via the following method:

```
INPUT_STATE *getInputState(int inIndex)
```

If the input image should be updated as well, you may use:

```
INPUT_STATE *getUpdatedInputState(int inIndex)
```

which will never return `CONNECTED_BUT_NEEDS_UPDATE`, since it will update the image properties if an update is required.

The `Module` class provides a method to handle the getting of updated input images.

```
PagedImage *getUpdatedInputImage(int i, bool getReal=false)
```

This is a convenience method for accessing the input image at index *`i`*. If there is any possibility to get a valid and accessible input image, this method will return a pointer to its `PagedImage`, otherwise `NULL` is returned.

3.1.11. Configuring Image Processing Behavior of the Module

The `Module` class offers some further methods to control image processing behavior. The following sections describe these features.

3.1.11.1. Inplace Image Processing

In some image processing algorithms the input and output pages have the same extent and data type. Hence, the algorithms might only need one buffer which is input *and* result (i.e. output) at the same time instead of having different buffers for the input and the output pages. Typical algorithms are e.g lookup, thresholding or arithmetic operations. You can instruct the ML to use only one buffer by

calling the `setOutputImageInplace(int outIndex=0, int inIndex=0)` method, because that avoids unnecessary buffer allocating and memory copying. Furthermore, the CPU does not need to switch between different memory areas which improves prefetching. The following methods are available to enable inplace operation for the `calculateOutputSubImage()` method:

```
//! Set optimization flag: If calculating a page in calculateOutputSubImage()
//! the output image page of output outIndex shall use the same
//! memory as the input page of input inIndex. So less allocations occur
//! and the read and written buffer are identical. Usually only useful for
//! pixel operations or algorithms which do not modify the image data.
//! Setting inIndex = -1 disables inplace optimization for the given outputIndex.

protected: void setOutputImageInplace(MLint outIndex=0, MLint inIndex=0);


//! Clear optimization flag: output page of output and input tile shall
//! use different memory buffers in calculateOutputSubImage().
//! This is an equivalent to setOutputImageInplace(outIndex, -1).

protected: void unsetOutputImageInplace(MLint outIndex=0);

//! Return optimization flag: Return index of input image whose input tile
//! is used also as output page for output outIndex in calculateOutputSubImage()
//! (instead of allocating its own memory). If inplace calculation is off
//! then -1 is returned.

public:    MLint getOutputImageInplace(MLint outIndex=0) const;
```

Note

- This mode is normally configured in the constructor but it can also be changed in the `handleNotification()` method. It is not recommended to change it in any `calc*()`method.
- The module cannot request the input image as a memory image by using `PagedImage::setInputSubImageUseMemoryImage()` if inplacing is activated. The ML will post errors in this case.
- The module still calls the `calculateOutputSubImage()` method with the same parameters as for non-inplace operation. However, the data pointers of the passed input and output subimages will point to the same memory area for the inplaced input and outputs. This may help to implement the algorithm more efficiently. However, it also needs to be considered that read and written buffers are the same for writing operations.

3.1.11.2. Bypassing Image Data

Some modules only change image properties, but do not modify actual image data. Examples of such algorithms are the `Switch` or the `Bypass` modules which only propagate data. Nor does the `ImagePropertyConvert` modify the image data when using its default behavior.

In this case, it is useful to avoid pages being processed by the module or being cached at the module's output. This reduces the amount of memory copies and the number of pages stored in the ML cache, i.e., the memory load of the application using these modules is reduced.

This feature can be configured by the following two methods (similar to the `setOutputImageInplace`() method) :

```
//! Sets the input image whose pages can also be used instead of output pages
//! to avoid recalculations. Setting an inIndex of -1 disables bypassing
//! (which is the default).
//! Bypassing require image (data) content, image extent, page extent and
//! voxel data type ro remain unchanged, or errors will occur.

protected: void setBypass(MLint outIndex=0, MLint inIndex=0);


//! Returns the currently bypass index or -1 if bypassing is disabled (default).
//! Bypassing require image (data) content, image extent, page extent and
//! voxel data type to remain unchanged, or errors will occur.
```

```
public:    MLint getBypass(MLint outIndex=0) const;
```

Note

- This option is not available in MeVisLab versions previous to 1.6 or in ML versions previous to 1.7.59.19.76.
- This mode is normally configured in the constructor but can also be changed in the `handleNotification()` method. It is not recommended to change it in any `calc*()` method.
- The module must still implement `calculateOutputSubImage` to calculate output pages, because the ML core cannot use bypassing in all situations. This can easily be done by activating the inplace mode and implementing `calculateOutputSubImage()` as an empty method.
- The module must not change the extent, voxel type or page extent of the image, because pages connected to the input image must have exactly the same memory layout as the pages calculated by the module. So do not modify any of these image properties in the `calculateOutputImageProperties()` method when you have enabled bypassing. If you do, the ML will post errors.

3.1.11.3. Multithreading: Processing Image Data in Parallel

The ML supports multithreading, i.e., it can perform image processing tasks in parallel if supported by the module's algorithm. Currently, only the `calculateOutputSubImage()` method of `Module` (or its overloaded method) is called in parallel. The following `Module` method and enumerator values are used to activate parallel computation:

```
//! Pass any THREAD_SUPPORT mode to decide whether and what type of multithreading
//! is supported by this module. See THREAD_SUPPORT for possible modes.

void setThreadSupport(THREAD_SUPPORT supportMode);


//! Enumerator deciding whether and which type of multithreading
//! is supported by this module.

enum THREAD_SUPPORT {

  //! The module is not thread safe at all.
  NO_THREAD_SUPPORT,

  //! calculateOutputSubImage is thread-safe for scalar voxel types.
  ML_CALCULATE_OUTPUTSUBIMAGE_ON_STD_TYPES,

  //! calculateOutputSubImage is thread-safe for all voxel types.
  ML_CALCULATE_OUTPUTSUBIMAGE_ON_ALL_TYPES,
};
```

Note

- This option is not available with enumeration values in MeVisLab versions previous to 1.6 or in ML versions previous to 1.7.59.19.76. They only provide enabling or disabling multithreading with 1 or 0 as parameters for images with standard (scalar) voxel types.
- This mode is normally configured in the constructor but can also be changed in the `handleNotification()` method. It is not recommended to change it in any `calc*()`method.

Important

Since multithreading errors are often difficult to debug, it must be made sure that algorithms are really thread-safe before the multithreaded execution of `calculateOutputSubImage()` is enabled.

To ensure thread-safe operations, it must be possible to execute many parallel versions of the algorithm without modifying shared data. Local variables, for example, are normally thread-safe, because they are stored in the local stack of the concerning thread. If parallel access to shared objects is required, special synchronization mechanisms must be used. The ML makes use of the `boost::thread` libary and provides simple wrappers in the header files `mlThread.h` , `mlMutex.h` , and `mlBarrier.h` as well as Section 3.1.11.3.1, “How to Implement Thread-Safe Code Fragments” for more information.

An algorithm (or to be more precise: `calculateOutputSubImage()`) is **not** thread-safe

- when it is not reentrant.
- when non-local variables are written without synchronization like mutex locking (see Section 3.1.11.3.1, “How to Implement Thread-Safe Code Fragments”).
- when any stream, debug or other console output is used; so do not use methods like `std::cout`, `std::cerr` or `printf`. It is safe to use `mlDebug`, `mlWarning`, `mlError` and `mlInfo`.
- when fields are accessed.
- when `getTile()` methods are called from within `calculateOutputSubImage()`. This is also `true` for `VirtualVolume` classes, because they use `getTile` internally.

In most (but not all!) cases, it is legal to modify and use the following objects in the implementation of `calculateOutputSubImage`:

- Non-static local objects of the function if they are marked as re-entrant classes (e.g., `ImageVector`, `SubImageBox` , `Vector2`, ..., `quaternions`, etc.),
- functions and methods if they do not modify data like constant get functions, read access to members, etc.,
- the input and output subimages passed to `calculateOutputSubImage` because they are thread-local objects (with the exception of input buffers using `MemoryImage`),
- all methods of input and output `SubImage` and `TSubImage` objects passed as `calculateOutputSubImage` parameters.

The following two sections discuss strategies of how to implement thread-safe code.

3.1.11.3.1. How to Implement Thread-Safe Code Fragments

In some cases, it might be useful to modify objects from within the `calculateOutputSubImage` function although it is called in parallel by the ML. This, for example, happens when statistical values are summed up from all pages and composed in members of the class. The most typical solution to this problem is to protect a code fragment against parallel execution with a so-called *mutex* implemented as a member of your class. Include `mlMutex.h` for such code.

```
#include "mlMutex.h"

ML_START_NAMESPACE

class ML_EXAMPLE_PROJECT_EXPORT ExampleModule : public Module
{
    // ...

private:

  //! The mutual exclusion object to protect a code fragment.
  Mutex _mutex;

  // The member or object to be protected against parallel modification.
  int   _myMember;

  //  ...
};


template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
```

```
                                    int outIndex,
                                    TSubImage<DATATYPE> *inSubImg)
{
  double voxVal = 0;

  // TODO: Here the loop calculates "voxVal"...

  {
    Lock(_mutex);
    // This area is protected against parallel execution. The area between
    // lock() and unlock() is entered only by one thread at once; another thread
    // will not pass lock() until the current thread has passed unlock().

    _myMember += voxVal;

  }

  //  ...
}

ML_END_NAMESPACE
```

Note

- The class `Mutex` was not available in MeVisLab versions previous to 1.6 or in ML versions previous to 1.7.59.19.76; the corresponding class was `mlCriticalSection`. Please refer to the documentation in the ML class reference for details.
- The mutex class provided by the ML allows the current thread to reenter the same section recursively, and counts the number of (un)lock operations. Careful use of this behavior is strongly recommended because mutex classes from other libraries might handle this differently.
- Protecting code fragments with `Mutex.lock()` and `Mutex.unlock()` is often time-critical. Hence, information should be collected in local variables (especially from inner loops, like *`voxVal`* in the above example) and the result should be written into shared members only once in a protected region (typically at the end of the function).
- The ML currently provides only mutex locking as a synchronization mechanism for multithreading although it is not the solution to all synchronizing/protection problems. It is recommended not to use multithreading when more complex mechanisms are needed.
- It is also recommended to be familiar with multithreaded programming before using it, because errors in that area tend to be hard to find and difficult to debug. For safety reasons, do not enable multithreading if there are any doubts.
- Since thread management requires overhead, it is recommended to test performance after activating multithreading to make sure that execution is really faster.

3.1.11.4. Processing Images of Registered Voxel Types

The ML supports processing of images with non-scalar and user-registered voxel types. See Chapter 7, *Registered Voxel Data Types* for detailed information on activation. In the default setup of an ML module, this feature is disabled and must be activated by the programmer when needed:

```
enum PERMITTED_TYPES {

  //! Allows only scalar voxel types, the default.
  ONLY_SCALAR_TYPES,

  //! Enables all scalar voxel types and a default set
  //! of extended voxel types like complex numbers and
  //! some vector and matrix types.
  ONLY_DEFAULT_TYPES,

  //! Enables all voxel types registered for the ML.
  ALL_REGISTERED_TYPES
};
```

```
//! Specifies which types this module supports. Default
//! is ONLY_SCALAR_TYPES.

void setVoxelDataTypeSupport(PERMITTED_TYPES permTypes);
```

Note

- Multithreading of registered voxel types is not available in MeVisLab versions previous to 1.6 or in ML versions previous to 1.7.59.19.76.
- This mode is normally configured in the constructor but can also be changed in the `handleNotification()` method. It is not recommended to change it in any `calc*()`method.

Important

Using registered voxel types in multithreaded modules requires additional care by the programmer. See Section 3.1.11.3, “Multithreading: Processing Image Data in Parallel” for details.

3.1.12. Explicit Image Data Requests from Module Inputs

Sometimes it might be useful to explicitly request image data from a module input. The `Module` provides some functions to do so. In such functions, a data type and a subimage from the input can be specified to get an explicit copy of that region in memory. This is also permitted in `calculateOutputSubImage()`, because a copy is needed for extraordinary image requests (which, however, requires multithreading for that module to be disabled). Note that the following functions are also available in `PagedImage` objects that are returned by the `getUpdatedInputImage()` and `getInputImage()` methods, shown as second versions.

The following functions are available:

1.
```
static MLErrorCode Module::getTile(Module               *op,
                                   int                   outIndex,
                                   SubImageBox           loc,
                                   MLDataType            dataType,
                                   void                **data,
                                   const ScaleShiftData &scaleShiftData =
                                     ScaleShiftData());
or
     MLErrorCode PagedImage::getTile(SubImageBox            loc,
                                   MLDataType            dataType,
                                   void                **data,
                                   const ScaleShiftData  &scaleShiftData =
                                     ScaleShiftData());
```

   This function requests a subimage region *`loc`* from the image at output *`outIndex`* of module *`op`* . The data is stored into memory with type *`dataType`* and scaled with the settings specified in *`scaleShiftData`*. *`data`* is a `void*` pointer; and there are two cases to distinguish. First, if the `void*` pointer is `NULL`, the necessary memory for the subimage data is allocated and the `void*` pointer is set to the allocated memory address. Second, if the pointer is not `NULL`, the memory address is used to store the subimage data; the memory must be sufficiently large to avoid buffer overrun errors.

   If the memory is allocated by the `getTile` function, the memory needs to be released by `MLFree()` (see `freeTile()` below).

2.
```
static MLErrorCode Module::getTile(Module               *op,
                                   int                   outIndex,
                                   SubImage              *subImg,
                                   const ScaleShiftData &scaleShiftData =
                                     ScaleShiftData());
or
     MLErrorCode PagedImage::getTile(SubImage             &subImg,
                                   const ScaleShiftData &scaleShiftData =
                                     ScaleShiftData());
```

Generally, this function operates in the same way as the first version did. However, data type, data pointer and subimage region are retrieved from *subImg*.

3.
```
static MLErrorCode Module::getTile(Module               *op,
                                   int                   outIndex,
                                   SubImageBox           loc,
                                   MLDataType            dataType,
                                   MLMemoryBlockHandle  &memoryBlockHandle,
                                   const ScaleShiftData &scaleShiftData);
or
     MLErrorCode PagedImage::getTile(SubImageBox           loc,
                                   MLDataType            dataType,
                                   MLMemoryBlockHandle  &memoryBlockHandle,
                                   const ScaleShiftData &scaleShiftData);
```

This function generally also works in the same way as the first version. However, the data pointer is retrieved from *memoryBlockHandle* and the allocated subimage is inserted into the current cache tables.

4. Use the function `freeTile()` to release the memory allocated by `getTile()` functions. It is safe to pass `NULL` pointers to `freeTile()`:

```
static void Module::freeTile(void* data);
or
     void PagedImage::freeTile(void* data);
```

Important

Using one of the above functions requires the addressed module outputs or images to be up to date. To test and/or to update outputs, `Module::getUpdatedInputImage()` should be used. (See Section 3.1.10.1, “Checking Module Inputs for Validity”).

Example 3.1. Explicitly Requesting Image Data (as `double` Voxels) from a Module Input:

```
if (getUpdatedInputImage(inputNum) != NULL) {

  // Pass NULL pointer for automatic memory allocation when calling getTile().
  void *data=NULL;

  // Get unscaled double data from box with subImgCorner1 and subImgCorner2.
  const MLErrorCode localErr = getTile(getInOp(inputNum), getInOpIndex(inputNum),
                                       SubImageBox(subImgCorner1, subImgCorner2),
                                       MLdoubleType,
                                       &data,
                                       ScaleShiftData(1,0));

  // Test for general errors and for out of memory.

if (localErr != ML_RESULT_OK) {
    if (ML_NO_MEMORY == localErr) {
      mlError("TestOp::loadData", ML_NO_MEMORY) << "Out of Memory!";
    } else{
      mlError("TestOp::loadData", localErr) << "Could not get input image tile!";
    }
  } else {

    // Everything okay, we can use the data.
  }

  // Free the allocated data and reset pointer.
  freeTile(data);
  data = NULL;
}
```

3.1.13. Getting Single Voxel Values from Module Inputs

Sometimes it is useful to request single voxel values from a module input. This can easily be done by using the following `Module` function:

Example 3.2. How to Get a Single Voxel Value from an Image as a String

```
static std::string getVoxelValueAsString(Module *op, int outIdx, const ImageVector &pos,
                                         MLErrorCode *errCode=NULL,
                                         const std::string &errResult="");
```

The function returns the voxel value at position *pos* of output *outIdx* of the module op as a standard string. When an error occurs, *errResult* is returned instead of the voxel value. *errCode* can be passed as NULL (the default). Otherwise, errors are reported in **errCode* or ML_RESULT_OK is set. If the requested voxel position is out of the image range, an empty string ("") is returned and **result* is set to ML_RESULT_OK.

Note

This function is a convenience function for single voxel access and uses getTile() calls internally, i.e., the function is not an efficient way to retrieve input image data. See Section 3.1.12, “Explicit Image Data Requests from Module Inputs” or Section 2.3.7, “ VirtualVolume ” when you need multiple or more efficient access methods.

3.1.14. Interrupting Page-Based Image Processing and Handling Errors

In a well designed ML module class derived from Module, there is normally no need to handle errors in calculateOutputSubImage() or calculateInputSubImageBox(), because invalid parameters are usually already handled or corrected in handleNotification() or the output image is invalidated in calculateOutputImageProperties() with outImage->setInvalid(). This is the usual way to ensure that further calls of other calc*()methods do not have to operate with incorrect settings.

These error handling options, however, do not cover all potential error sources. When a module reads data from a file in calculateOutputSubImage(), for example, a file IO error could occur. Since no calc*() method offers return values and an invalidating of the output image is too late, there is only the option to throw an exception. The following code fragment demonstrates how this can be implemented in all calc*() methods but calculateOutputImageProperties():

```
template <typename T>
void ExampleModule::calculateOutputSubImage(TSubImage<T> *outSubImg, int outIndex)
{
  MLErrorCode errCode = _loader->getTileFromFileIntoSubImg(*outSubImg);

  if (ML_RESULT_OK != errCode) {

    throw errCode; // Throw error to terminate loading process.
  }
}
```

Note

- The ML will return the thrown error code or a resulting one in the top-level getTile() command which caused this calculateOutputSubImage() call and will also post it to the ML error handler. Processed pages will be all or partially invalid.
- Throwing errors in calculateOutputSubImage() should currently only be used for failure recovery. If possible, try to handle or correct incorrect parameters in handleNotification() or to invalidate the output image in calculateOutputImageProperties() to avoid errors before they can occur in other calc*()routines.

3.1.15. Testing for Interruptions During Calculations

In some algorithms, it might be useful to check whether a stop button has been pressed to provide the option to terminate long calculations. The function

```
//! Checks if a notify button was pressed (outside of normal notification)
//! It returns the notify field or NULL if nothing was pressed. Note that
//! more than one field may have been notified; so use a loop until NULL is
//! returned to be sure that all NotifyFields have been checked.

Field *Module::getPressedNotifyField();
```

performs such a check on notify fields. A corresponding field can be created in the constructor:

```
NotifyField *_stopButtonFld; // Header

_stopButtonFld = addNotify("stop"); // Implementation
```

The following function checks whether the button has been pressed during operation:

```
bool stopPressed()
{
  Field* field = NULL;

  do {

    field = getPressedNotifyField();

    if (field == _stopButtonFld) {
      return true;
    }
  } while (field != NULL);

  return false;
}
```

An alternative way to check if processing should be terminated is to call

```
bool Module::shouldTerminate();
```

This method returns true if any button has been pressed that was marked with `globalStop = true` in the MDL definition (even if it belongs to another modules), or if the stop button in the lower right corner of the IDE main window was pressed.

Note

- Both methods for break checking should not be employed inside of paged image processing calculations, e.g., inside `calculateOutputSubImage`. These functions may get called from other threads and the global stop mechanism represented by `shouldTerminate` is applied to the image processing loop anyway.

  Only use these methods if you start your own calculation loop from `handleNotification`.

- Checking for interruptions is system-dependent and requires the application using the ML to set up the function `Host::setBreakCheckCB()` correctly. This is done correctly in MeVisLab (the typical context where the ML is used), but might not be possible when using the ML in standalone programs. So be careful when developing an algorithm, and document in how far your algorithm requires this functionality.

- The check for interruptions set up by `Host::setBreakCheckCB()` might be expensive and might degrade the performance of the calling algorithm when it is called too often.

3.1.16. Adapting Page Extents

Normally, a programmer does not need to not care about the extent of pages, because import modules such as `ImageLoad` normally set it up appropriately.

However, some modules change the extent of images or even generate new images that require the calculation of new page extents. An appropriate extent of pages depends on many parameters, e.g., on the dimension of an image, its extent, whether it uses colors, the types of algorithms processing it, the number of processors or threads, the memory size or even whether it is processed on a 32 or 64 bit system. The following convenience function implements a heuristic to provide an appropriate page extent:

```
//! Adapt page extent. Arguments are:
//! - pageExt:    Suggested page extent (e.g., of input image), overwritten
//!               by new page extent
//! - imgType:    Data type of output image
//! - newImgExt:  Extent of output image
//! - oldImgExt:  Extent of input image
//! - pageUnit:   Page extent must be a multiple of this, or
/?!               ImageVector(0) if do not care
//! - minPageExt: Minimum page extent, or ImageVector(0) if do not care
//! - maxPageExt: Maximum page extent, or ImageVector(0) if do not care

static void ModuleTools::adaptPageExtent(ImageVector &pageExt,
                                         MLDataType imgType,
                                         const ImageVector &newImgExt,
                                         const ImageVector &oldImgExt,
                                         const ImageVector &pageUnit   = ImageVector(0),
                                         const ImageVector &minPageExt = ImageVector(0),
                                         const ImageVector &maxPageExt = ImageVector(0));
```

Note

The correct position to call the convenience function is inside the method `calculateOutputImageProperties()`, because all the properties of output images are specified there.

3.1.17. Processing Input Images Sequentially

Certain algorithms are hard to implement with the image processing methods presented so far. These are algorithms that are applied to an image to only "extract" information instead of modifying the image data. Such algorithms need a special call due to the fact that the extraction of information is not triggered page-wise by any consuming module.

For this purpose, the following special `Module` method can be called:

```
MLErrorCode processAllPages(int outIndex = -1)
```

This method processes all pages of an image and allows for an easy implementation of page-based image processing algorithms on entire images. Internally, all pages of the output image with index `outIndex` are requested as from a connected (consuming) module.

A common image processing is executed with the following deviations:

- If *`outIndex`* is -1, a temporary output image with the same negative index -1 is created and `calculateOutputImageProperties()` is called with an *`outIndex`* of -1 and the temporary output image as *`outImage`*. By checking for outIndex == -1, it can be detected if the call was initiated by `processAllPages()` and the properties of the temporary output image can be adjusted as desired. By default, the temporary output image has the properties of the first input image (at inputIndex == 0).

- If *`outIndex`* is -1, as described in Section 3.1.9, “Implementing calculateOutputSubImage()”, the output pages must **not** be written since no data is allocated for them to improve performance for pure input scanning algorithms.

- The output index *`outIndex`* is passed to `calculateOutputSubImage()` and `calculateInputSubImageBox()`, even if it is -1 (see Section 3.1.9, “Implementing calculateOutputSubImage()” and Section 3.1.7, “Implementing calculateInputSubImageBox()”). By checking if the value is -1, you know that the output must not be written and that the call comes from `processAllPages()`.

The return value is `ML_RESULT_OK` on a successful operation, otherwise a code describing the error is returned.

As in common page-based image processing, all pages of the input image(s) are requested from the input(s) in order to process the (possibly not existing) output page. Thus multiple inputs can be processed simultaneously with almost the same concept as it is done in common page processing. If

only one input is to be scanned and if others are to be ignored, simply request empty page boxes for those (see Section 3.1.7, “Implementing calculateInputSubImageBox()”).

The following example demonstrates how to calculate the average of all voxels from input 0 whose corresponding voxels from input 1 are not zero. Input 2 will be ignored:

Example 3.3. Average Calculation of Masked Voxels in a 3-Input Module

```
// ********** HEADER FILE:

#include "mlModuleIncludes.h"


ML_START_NAMESPACE

class ExampleModule : public Module
{

protected:

  ExampleModule();

  virtual void handleNotification(Field *f);
  virtual SubImageBox calculateInputSubImageBox(int inIndex,
                                     SubImageBox &outBox,
                                     int /*outIndex*/);
  virtual void calculateOutputSubImage(SubImage *outSubImg,
                               int outIndex,
                               SubImage *inSubImgs);
  template <typename DATATYPE>
  void calculateOutputSubImage(TSubImage<DATATYPE> * /*outSubImg*/,
                       int outIndex,
                       TSubImage<DATATYPE> *inSubImg0,
                       TSubImage<DATATYPE> *inSubImg1,
                       TSubImage<DATATYPE> * /*inSubImg2*/);

private:

  NotifyField *_processPagesFld;
  long double  _voxelSum;
  long int     _voxelNum;

  ML_MODULE_CLASS_HEADER(ExampleModule);
};

ML_END_NAMESPACE
```

```
// ********** SOURCE FILE:
ML_START_NAMESPACE

ML_MODULE_CLASS_SOURCE(ExampleModule, Module);

ExampleModule::ExampleModule(): Module(3,1)
{
  _processPagesFld = addNotify("ProcessPages");
}
```

```
void ExampleModule::handleNotification(Field *f)
{
  if (f == _processPagesFld) {
    _voxelSum = 0;
    _voxelNum = 0;

    processAllPages(-1);

    if (_voxelNum != 0) {
      mlDebug("Masked Average:" << _voxelSum/_voxelNum);
    } else {
      mlDebug("No masked voxels");
    }
  }
}
```

```
SubImageBox ExampleModule::calculateInputSubImageBox(int inIndex,
                                          SubImageBox &outBox,
                                          int /*outIndex*/)
{
  // Request page boxes from inputs 0 and 1 and get empty
  // region from input 2.
```

```
  if (inIndex == 2){

    return SubImageBox();
  } else {
    return outBox;
  }
}
```

```
// Implement the calls of the right template code for the
// current image data type.

ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_3_SCALAR_TYPES_CPP(ExampleModule);

template <typename DATATYPE>
void ExampleModule::calculateOutputSubImage(TSubImage<DATATYPE> * /*outSubImg*/,
                                    int outIndex,
                                    TSubImage<DATATYPE> *inSubImg0,
                                    TSubImage<DATATYPE> *inSubImg1,
                                    TSubImage<DATATYPE> * /*inSubImg2*/)
{
  // Get valid page box clamped to valid image regions. Then
  // scan all voxels in box.
  SubImageBox box = inSubImg0->getValidRegion();

  ImageVector p = box.v1;
  for (p.u = box.v1.u;  p.u <= box.v2.u;  ++p.u) {
    for (p.t = box.v1.t;  p.t <= box.v2.t;  ++p.t) {
      for (p.c = box.v1.c;  p.c <= box.v2.c;  ++p.c) {
        for (p.z = box.v1.z;  p.z <= box.v2.z;  ++p.z) {
          for (p.y = box.v1.y;  p.y <= box.v2.y;  ++p.y) {
            p.x = box.v1.x;
            DATATYPE* i0P = inSubImg0->getImagePointer(p);
            DATATYPE* i1P = inSubImg1->getImagePointer(p);

            for (;  p.x <= box.v2.x;  ++p.x){

              if (*i1P != 0) {

                // Sum up masked voxels
                ++_voxelNum;
                _voxelSum += *i0P;
              }

              // Move input pointers forward.
              ++i0P; ++i1P;
            }
          }
        }
      }
    }
  }
}

ML_END_NAMESPACE
```

3.1.18. Traps and Pitfalls in Classes Derived from Module

This section discusses typical errors in programming image processing filters derived from `Module`:

Typical errors are

- to forget to implement `ML_MODULE_CLASS_SOURCE`, `ML_MODULE_CLASS_HEADER` or to call the `initClass()` function (mostly in the Init file of the dll/shared object). It registers the class in the runtime type system of the ML. In MeVisLab, also a .def file with the `MLModule` entry and the correct DLL tag must exist.

  This causes e.g., MeVisLab to not being able to detect or create the module on a network.

- to forget to overload the virtual method `activateAttachments()` in your module if non-field members in the class depend on field settings.

  This leads to incorrectly restored module networks, e.g., in MeVisLab. See Section 3.1.3, “Module Persistence and Overloading activateAttachments()” for details.

- to forget to suppress calls of the method `handleNotification()` while fields in the constructor are added and initialized.

  This causes calls of `handleNotification()` with unexpected results or crashes during module initialization. Use the methods `handleNotificationOff()` and `handleNotificationOn()` around the initialization area of fields in the constructor.

- to forget to connect input connector or other fields with the output connector fields if an automatic update of the output image is desired when these fields change.

  This often leads to output images that are not or only partially up to date or that do not update correctly on parameter/field changes.

- to change the number of inputs in the superclass call of Module (e.g., `MyClass(...) : Module(numInputs, numOutputs)`) and to forget to change the `ML_CALCULATE_OUTPUTSUBIMAGE` macro **and** the parameters of the `calculateOutputSubImage()`template.

  This problem is not detected by some compilers and leads to empty or missing implementations of `calculateOutputSubImage()`.

- to enable the thread support without an explicit check whether `calculateOutputSubImage()` is really thread-safe.

  See Section 3.1.11.3, “Multithreading: Processing Image Data in Parallel” for details.

- to change the properties of output images outside the `calculateOutputImageProperties()` method or even from inside other `calc*` methods.

  See Section 3.1.6, “Implementing calculateOutputImageProperties()” for details.

- to forget to check the validity of the input images or connectors when accessing inputs in `handleNotification()`.

  Use `getUpdatedInputImage()` to check and get the input image correctly. Note that the ML guarantees valid input images in all `calc*` methods. This permits the access of these images directly with `getInputImage(idx)`without further validity checks. See Section 3.1.10.1, “Checking Module Inputs for Validity” and Section 3.1.10, “Handling Disconnected or Invalid Inputs by Overloading handleInput()” for more details.

- to forget to clip the extent of the processed output page in `calculateOutputSubImage()` against the extent of the output image.

  Since pages can reach outside the image, unused regions are processed and possibly read from the input buffers. Although it is not an error to fill regions of the output page that reach outside the image, it is useless and adversely affects performance. The problem can simply be solved by clipping the region of the processed output page against the output image region:

```
const SubImageBox boxToProcess = outSubImg->getValidRegion();
```

- to forget that a `SubImageBox` has two corners *v1* and *v2* which **both** are part of the described region. Empty regions are denoted by any component in *v2* which is smaller than the corresponding one in *v1*.

  Hence, to process the region **box** "<=" comparisons are needed in the loops over all dimensions:

```
ImageVector p = box.v1;

for (p.u = box.v1.u;  p.u <= box.v2.u;  p.u++) {
  for (p.t = box.v1.t;  p.t <= box.v2.t;  p.t++) {
    for (p.c = box.v1.c;  p.c <= box.v2.c;  p.c++) {
      for (p.z = box.v1.z;  p.z <= box.v2.z;  p.z++) {
        for (p.y = box.v1.y;  p.y <= box.v2.y;  p.y++) {
          for (p.x = box.v1.x;  p.x <= box.v2.x;  p.x++) {
            // . . .
          }
        }
```

```
      }
    }
  }
}
```

- to forget that the default behavior of a `Module` class is to pass input data to `calculateOutputSubImage()` which is of the type of the **output** image, even if the connected input image has another voxel type. Hence, the input data might be cast implicitly. This typically simplifies module programming and created code, because the type of input and output voxels are identical and only one template argument is needed for `calculateOutputSubImage()`. Note that the default behavior of a `Module` class is for the output image to inherit the data type of the input image which minimizes data conversions and ensures that all modules process the same data type by default.

  See Section 3.1.6, “Implementing calculateOutputImageProperties()” for details if you need to have differently typed input and output buffers.

- to forget that a dynamic number of input subimages implemented with a `ML_CALCULATE_OUTPUTSUBIMAGE_NUM_INPUTS_*N*` macro requires the implementation of the template function `calculateOutputSubImage` with a `T` at its end. Otherwise, internal ambiguities with other inherited `Module` methods appear.

Chapter 4. Image Processing Concepts

Chapter Objectives

By reading this chapter you will understand how image processing in the ML is organized and you will be able to differentiate your image processing algorithms between

- voxel-based or page-based (see Section 4.2.1, “Page-Based Concept” and Section 4.2.2, “Voxel-Based Concept ”),
- slice-based (see Section 4.2.3, “Slice-Based Concept”),
- kernel-based (see Kernel Progamming),
- partially global approaches like
  - random access (see Section 4.3.1, “Random Access Concept (Tile Requesting)”),
  - sequential access (see Section 4.3.2, “Sequential Image Processing Concept”) and
  - virtual volume (see Section 4.3.3, “`VirtualVolume` Concept”)
- global image processing approaches like
  - temporary global (see Section 4.4.1, “Temporary Global Concept”)
  - global (see Section 4.4.2, “Global Image Processing Concept”)
  - `BitImage` (see Section 4.4.3, “`BitImage` Concept”)
  - `MemoryImage` (see Section 4.4.4, “`MemoryImage` Concept”)
- mixed modules (see Section 4.5, “Miscellaneous Modules”)

and you will be able to select the appropriate ways to implement these processing algorithms in the ML.

4.1. Page Calculation in the ML

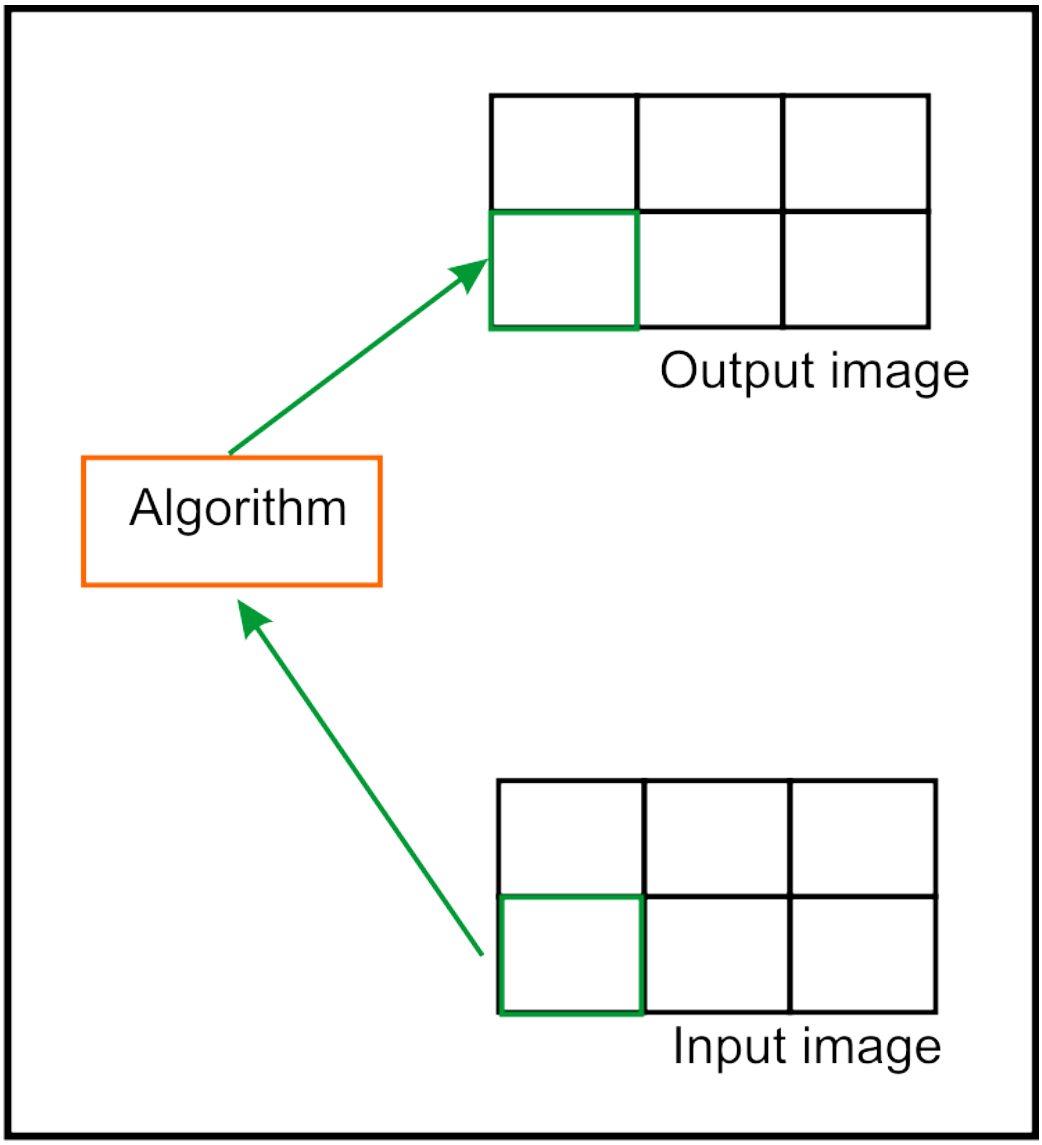
Page-based image processing is one of the key concepts in the ML, because filtering and analyzing images easily fails when images do not fit into memory.

4.2. Page-Based Approaches

4.2.1. Page-Based Concept

Used if voxel coordinates are not necessary and voxel operations are local. (LUT, windowing, some color model changes, thresholding, inversion, arithmetics on voxel data, etc.).

Figure 4.1. Page-Based Concept



Advantages:

- Fast image data access by pointer incrementation
- Short implementation

Disadvantages:

- Voxel coordinates are not directly available
- Neighbour voxels are only available with precautions
- Not very useful for complicated algorithms
- Precautions necessary because pages could reach outside the image, i.e., voxels outside the image might be processed.

Example 4.1. Implementing a Page-Based Algorithm

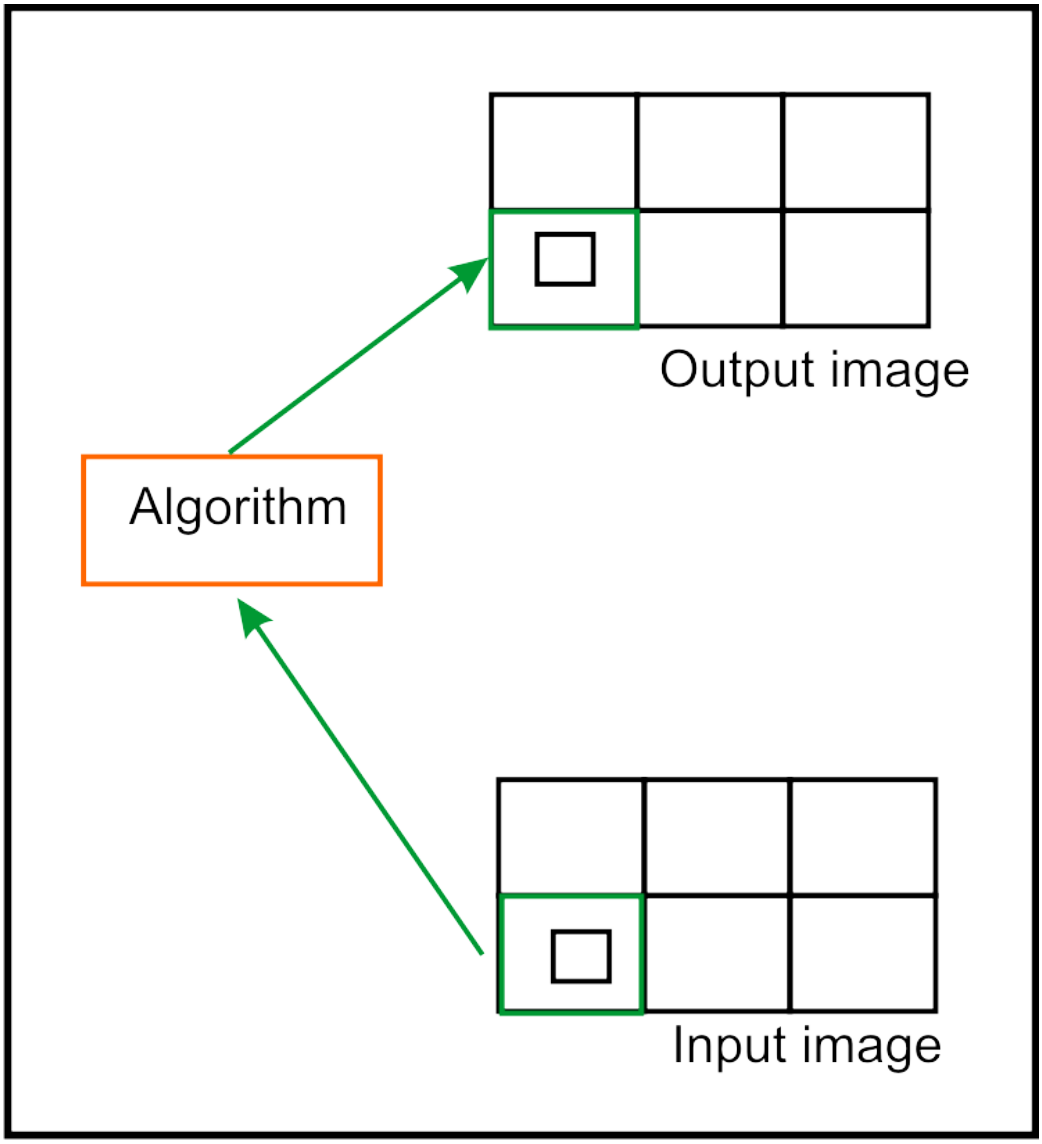
```
template <typename DATATYPE>
void AddExample::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                int              /*outIndex*/,
                                TSubImage<DATATYPE> *inSubImg1,
                                TSubImage<DATATYPE> *inSubImg2)
{
  // Get pointers to memory buffers of input and output subimage.
  DATATYPE* outSubImgVP_beg = outSubImg->getImagePointer(outSubImg->getBox().v1);
  DATATYPE* outSubImgVP_end = outSubImg->getImagePointer(outSubImg->getBox().v2);
  DATATYPE* inSubImg1VP     = inSubImg1->getImagePointer(inSubImg1->getBox().v1);
  DATATYPE* inSubImg2VP     = inSubImg2->getImagePointer(inSubImg2->getBox().v1);

  // Loop over all voxels in memory buffers even if pages reache outside the image.
  for (DATATYPE* outSubImgVP = outSubImgVP_beg;
       outSubImgVP <= outSubImgVP_end;
       outSubImgVP++, inSubImg1VP++, inSubImg2VP++)
  {
    (*outSubImgVP) = (*inSubImg1VP)  + (*inSubImg2VP);
  }
}
```

4.2.2. Voxel-Based Concept

Useful for all pixel-based algorithms already mentioned in Section 4.2.1, “Page-Based Concept” (LUT, windowing, some color model changes, thresholding, inversion, arithmetics) or if voxel coordinates are essential and operations are local (rasterization of implicit Objects, `SubImage`, etc.), e.g. `mlAddExampleOp`.

Figure 4.2. Voxel-Based Concept



Advantages:

- Fast access by 6 nested loops and pointer incrementation in inner loop
- Voxel coordinates are available
- Conceptually good implementation, recommended for page processing

Disadvantages:

- Neighbor voxels only available with precautions
- Not very useful for advanced algorithms

Example 4.2. Implementing a Voxel-Based Algorithm

```
template <typename DATATYPE>
  void PosExample::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg, int outIndex,
                                   TSubImage<DATATYPE> *inSubImg)
{
  // Get extent of output image and clamp the extent of the box of outSubImg
  // against
  // it to be sure that no voxels outside the image are processed.

  SubImageBox box    = outSubImg->getValidRegion();

  // Iterate over all valid voxels of inSubImg and outSubImg.
  ImageVector p = box.v1;
  for (p.u = box.v1.u;  p.u <= box.v2.u;  ++p.u) {
    for (p.t = box.v1.t;  p.t <= box.v2.t;  ++p.t) {
      for (p.c = box.v1.c;  p.c <= box.v2.c;  ++p.c) {
        for (p.z = box.v1.z;  p.z <= box.v2.z;  ++p.z) {
          for (p.y = box.v1.y;  p.y <= box.v2.y;  ++p.y) {

            // Get/Set position of row starts as pointers to memory
            // positions in inSubImg and outSubImg buffers.

            p.x = box.v1.x;
            DATATYPE* iP = inSubImg ->getImagePointer(p);
            DATATYPE* oP = outSubImg->getImagePointer(p);

            // Process all voxels in row with pointers. Be sure to
            // include last voxel in row with "<= box.v2.x", because
            // v2 is still part of box region.

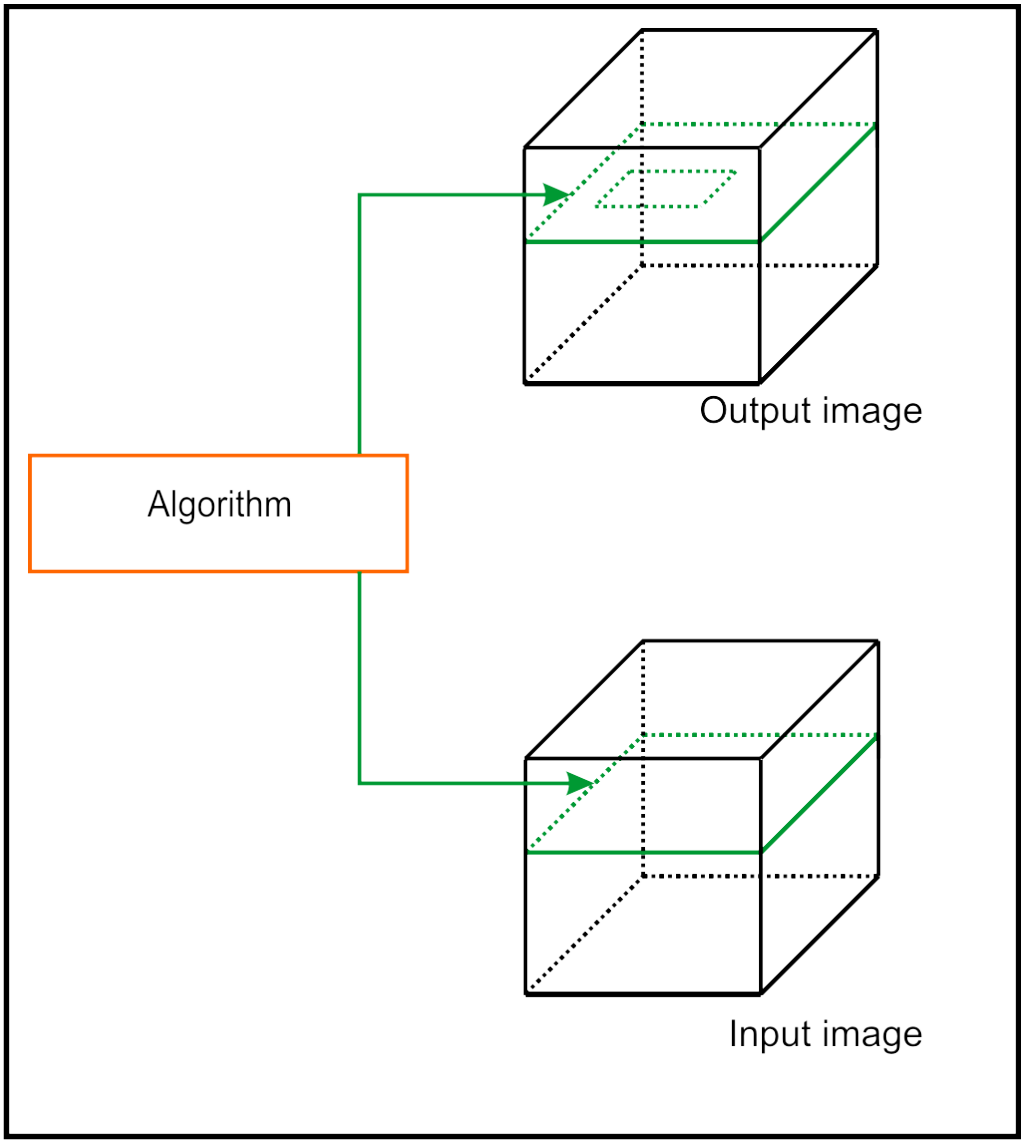
            for (;  p.x <= box.v2.x;  ++p.x)
            {
              *oP = calcFromIp(p.x, *iP); // Calculate voxel from position & input
              ++iP; ++oP;                 // Move input and output pointer forward
            }
          }
        }
      }
    }
  }
}
```

See Section 3.1.9, “Implementing calculateOutputSubImage()”, Section 7.2.3, “Examples with Registered Voxel Types” and programming examples released with MeVisLab for further examples.

4.2.3. Slice-Based Concept

Useful for arbitrary 2D algorithms. The page extent is set to slice extent, or for calculations of an output page, the entire input slice is requested in `calculateInputSubImageBox()`.

Figure 4.3. Slice-Based Concept



Advantages:

- Very fast random access (with getValue/setValue or like page-based or voxel-based concept)
- Easy to implement
- Paging still works fine if x and y extents are not too large

Disadvantages:

- PageExt == SliceExt: can easily degenerate and become very expensive (e.g., on large mammograms or satellite images), also page extent is propagated to appended images
- InputTile == SliceExt, output page is normal: many slice requests become necessary to compose the output slice

Consider whether e.g., the `VirtualVolume` or a kernel-based concept could replace this concept to avoid these disadvantages.

Example 4.3. Implementing a Slice-Based Algorithm

```
void SliceFilter::calculateOutputImageProperties(int /*outIndex*/, PagedImage* outImage)
{
  // Set extent of pages in z, c, t and u dimension to 1.
  // Thus only axial slices will be calculated by the module.
  // Avoid too small pages.

  ImageVector pExt = getInputImage(0)->getPageExtent();
  if (pExt.x < 64){ pExt.x = 64; }
  if (pExt.y < 64){ pExt.y = 64; }
  outImage->setPageExtent(ImageVector(pExt.x, pExt.y, 1,1,1,1));
}

SubImageBox SliceFilter::calculateInputSubImageBox(int /*inIndex*/,
                                        const SubImageBox& outSubImgBox,
                                        int /*outIndex*/)
{
  // Request slice with image x/y extent. All other
  // parameters are given by the page extent.

  SubImageBox inBox = outSubImgBox;
  inBox.v1.x = 0;
  inBox.v1.y = 0;
  inBox.v2.x = getInputImage(0)->getImageExtent().x-1;
  inBox.v2.y = getInputImage(0)->getImageExtent().y-1;
  return inBox;
}
```

4.2.4. Kernel-Based Concept

An important class of image filters is based on the so-called *kernel-based image filtering*. This class is used when a fixed region around a voxel is needed to calculate output voxel (edge detector operations, morphological operations, noise filters, smoothing, texture filters, etc.).

Advantages:

- Fast access to kernel range in 6D is possible with paging so it fits well into page concept
- Many algorithm categories can be implemented

Disadvantages:

- Base class is a bit more complex
- Image borders require consideration (supported by base classes, though)

See Kernel Progamming for more information.

4.3. Concepts for Partially Global Image Processing

4.3.1. Random Access Concept (Tile Requesting)

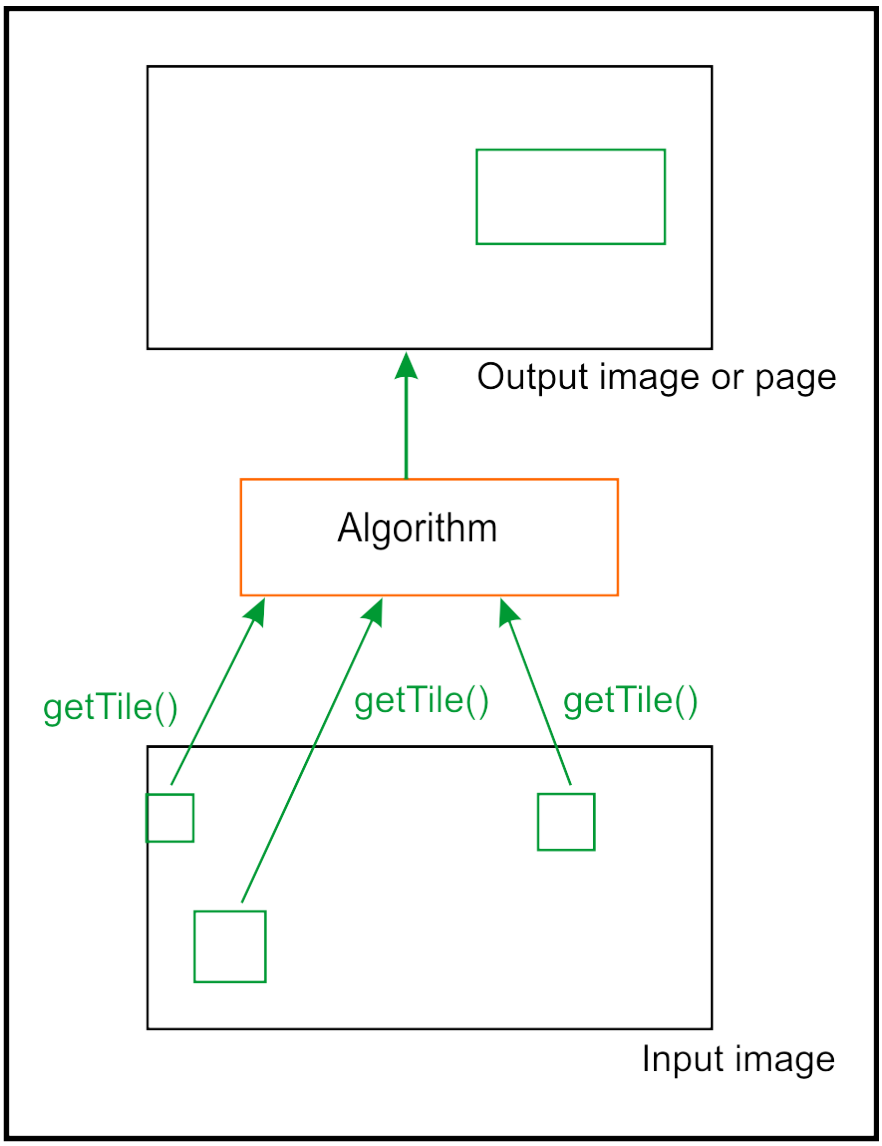
There are different ways to implement algorithms that need random image access.

One way is to use the "explicit image data request" concept to request arbitrary tiles from the input image and to manage them as data chunks This is often useful when explicit data needs to be passed to function calls or direct pointer access is needed. See Section 3.1.12, “Explicit Image Data Requests from Module Inputs” for more information.

Another way is to use the "virtual volume" concept. This concept is especially useful for accessing very large images where no direct pointer or memory access and `set`/`getValue` methods are sufficient. See

Section 4.3.3, “VirtualVolume Concept” for more information and Section 2.3.7, “ VirtualVolume ” for examples.

Figure 4.4. Random Access Concept



4.3.2. Sequential Image Processing Concept

There are different approaches to processing one or more input images sequentially. In order to process very large images that may not fit into memory, it is crucial to perform the processes step by step. Some algorithms can simply do this page-wise, and other algorithms need random access.

The most common approach is to use the `processAllPages` command available as a function in the class `Module` to force the processing of all pages via `calculateOutputSubImage()` calls. This concept is discussed in detail in Section 3.1.17, “Processing Input Images Sequentially” and is very similar to the implementation of a normal page-based module. The example calculates a masked average of all image voxels in a page-based manner.

The "virtual volume" concept is another concept often used. This concept provides random access to the managed image. Then it is easy to implement a normal loop to traverse all voxels or to use the `moveCursorXWrapAround()` function on a typed virtual volume to move a cursor over each voxel of the image, comparable to an iterator. See Section 4.3.3, “VirtualVolume Concept” for more information and Section 2.3.7, “ VirtualVolume ” for examples.

4.3.3. `VirtualVolume` Concept

The `VirtualVolume` and the `TVirtualVolume` classes manage an efficient voxel access to the output image of an input module or to a 'standalone' image. See Section 2.3.7, “ `VirtualVolume` ” for example code.

So it is possible to implement random access to a paged input image or to a pure virtual image without mapping more than a limited number of bytes. Pages of the input volume are mapped temporarily into memory when needed. If no input volume is specified, the pages are created and filled with a fill value. When the permitted memory size is exceeded, older mapped pages are removed. When pages are written, they are mapped until the virtual volume instance is removed or until they are explicitly cleared by the application. Virtual volumes can easily be accessed by using `setVoxel` and `getValue`. These kind of accesses are well-optimized code that might need between 9 (1D), 18 (3D) and 36 (6D) instructions per voxel if the page at the position is already mapped.

A cursor manager for moving the cursor with `moveCursor*` (forward) and `reverseMoveCursor*` (backward) is also available. About 5-9 instructions might be executed for these move methods. `setCursorValue` and `getCursorValue` provide voxel access. Good compilers and already mapped images might require about 5-7 instructions. So the cursor approach will probably be faster for data volumes with more than two dimensions.

All the virtual volume access calls can be executed with or without error handling (see last and default parameter of constructors). If *`areExceptionsOn`* is `true`, every access to the virtual volume is tested and if necessary, exceptions are thrown which can be caught by the code calling the virtual volume methods. Otherwise, most functions do not perform error handling.

Note

Exception handling versions are slower than versions with disabled exceptions. However, this is the only way to handle accesses safely.

Tip

This class is the recommended alternative for global image processing algorithms to using an actual global image (`MemoryImage`).

4.4. Global Image Processing Concepts

4.4.1. Temporary Global Concept

This concept has been designed for algorithms that require a single, very efficient random image access to calculate a result.

Important

If possible, try to avoid this approach!

It supports only limited image sizes which depend on the available memory!

Procedure:

- Read entire input image (e.g., when first voxel or page is requested or when the user starts the algorithm by pressing some button)
- Analyze the image and save all results
- Free allocated image buffer

- For page requests, the results are used to fill those pages

Advantages:

- Global algorithms are easy to implement, because all image data is directly available.
- Uses the memory only temporarily, i.e the memory is available again when the process has been finished.

Disadvantages:

- Results must be saved/buffered in data structures and must be reconverted into requested pages. Hence, additional structures are required.
- The image might not fit into memory.
- Even if only one voxel or a small page is requested, the entire volume must be processed.
- Danger of heap fragmentation.

4.4.2. Global Image Processing Concept

This concept is needed for time-critical random image access to calculate a result.

Important

If possible, try to avoid this approach!

It supports only limited image sizes which depend on the available memory!

Procedure:

- Read entire input image (e.g., when first voxel or page is requested or when the user starts the algorithm by pressing some button)
- Analyze the image and save all results
- Free allocated image buffer
- For page requests, the results are used to fill those pages.

Advantages:

- Easy and fast implementation.

Disadvantages:

- Degenerates the page-based, cached calculation process.
- Fails on images which do not fit into memory.
- Must map the entire image and blocks large memory areas for a long time. It cannot/should not be used in larger networks or applications because the "Out of Memory" state is easily reached.
- Danger of heap fragmentation.
- Usually the image must be passed page-based to the output which also requires additional implementation of page-based access to image.

Examples:

- See Section 4.2.3, “Slice-Based Concept”: The requested page has the size of the entire image.

- See Section 4.3.1, “Random Access Concept (Tile Requesting)”: The entire image is requested as a tile.

4.4.3. `BitImage` Concept

In the page-based image processing concept of the ML, Boolean data types are not available (nor are they planned).

The `BitImage` class can be used as an alternative option.

Advantages:

- Easy-to-use and compact image
- quite compact image although it is a global image
- relatively fast.

Disadvantages:

- Not paged, i.e., global; this, however, is not really problematic because only bits are stored

See Section 2.3.6, “ `BitImage` ” for a detailed overview.

4.4.4. `MemoryImage` Concept

Algorithms that need access to a whole non-paged memory-mapped image might use the `MemoryImage` approach for image processing (see Section 2.3.8, “ `MemoryImage` ”). Note that this breaks the page-based approach - nevertheless it is supported by the ML. It is integrated as a special member of the `PagedImage` in such a way that it can be handled in parallel or instead of a paged image (see Section 2.3.4, “ `PagedImage` ”).

4.5. Miscellaneous Modules

A set of miscellaneous module types can be considered, e.g.

- visualization modules showing an ML image in a viewer
- visualization modules creating ML images from their views
- converter from ML images to other information structures, such as object lists, histograms, model information or segmentations

The following examples give a basic idea of the different module types:

- ML Image -> Visualization

  Examples of such visualization modules are simple viewers which take the image data or the information derived from the image and show them on a display. This could include slice viewing (2D), volume or surface rendering (3D) or animated images in 2D/3D.

  MeVisLab also offers a set of specialized Open Inventor™ modules to accomplish these tasks. The modules get access to an ML image via an `SoSFMLImage` field. This gives access to the image data via `getTile()`.

  Examples in MeVisLab are `SoView2D`, `GlobalStat`, `SoGVRVolumeRenderer`.

- Visualization -> ML image

A typical example is a snapshot module creating ML images from (sequences of) image areas. Examples in MeVisLab are all viewers like `SoExaminerViewer` and the module `VoxelizeInventorScene`.

Chapter 5. Debugging and Error Handling

Chapter Objectives

The ML offers some special support for debugging, error handling, logging and exception handling:

- Section 5.1, “Printing Debug Information”
- Section 5.2, “Handling Errors”
- Section 5.3, “Registering Error Handlers”
- Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs”
- Section 5.5, “Tracing, Exception Handling and Checked Object Construction/Destruction”

5.1. Printing Debug Information

Debug printing is controllable in and by the ML and there is some material for selective debug printing. The required files are automatically included when the standard ML include file `mlModuleIncludes.h` . is used.

Controlling and Managing Debug Messages

The ML controls and manages debug (and other) messages by using the instance *`MLErrorOutput`* of the class `ErrorOutput` (see Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs”). It controls the debug outputs and the error handling system. However, this instance or class should not be used directly. It is recommended to use the `CoreControl` module which makes the important settings available (if it is possible by an e.g., application like MeVisLab. There, the debug printing can be enabled/disabled for the entire ML, and debugging can be enabled/disabled for certain classes by using environment variables or debug symbols.

Printing Debug Messages in the Source Code

In the source code e.g., of your project, usually one of the following macros generates the debug prints:

1. `mlDebug(STREAMOUTS)`(see mlDebug below - number 1)
2. `mlDebugPrint(STREAMOUTS)`(see mlDebugPrint below - number 3)
3. `mlDebugClass(CLASS_NAME, STREAMOUTS)`(see mlDebugClass below - number 4)
4. `mlDebugConst(ENV_VAR, STREAMOUTS)`(see mlDebugConst below - number: 2)
5. `mlDebugConditional(COND_SYM, STREAMOUTS)`(see mlDebugConditional below - number: 5)

Important

Each debug output is normally related to a debug symbol which must be enabled in the ML before the debug information can be printed.

Such a debug symbol can be defined as

1. an environment variable before the ML or the application is started,
2. as a debug symbol in the `CoreControl` module when used in an application such as MeVisLab,
3. or directly via programming in the global `MLErrorOutput` (see also Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs”) instance of the ML

The third option should not be used in normal code, but only in modules dedicated to debug control or diagnostics.

Important

To improve performance and reduce the amount of code, all debug macros are **not** compiled in release mode.

If debugging is enabled and the related debug symbol (or environment variable) for the macro is defined, any of the debug macros described below will send

- the file name,
- the time stamp,
- the line number,
- the debug symbol,
- and the passed parameter `STREAMOUTS`

to the global instance `MLErrorOutput` of the ML. This instance will send the above information to all registered instances (modules such as `Console`, `MLLogFile` and MeVisLab application consoles.

The ML provides the following macros for printing debug information:

1. `mlDebug(STREAMOUTS)`

   This macro prints the information given by `STREAMOUTS`. It requires the runtime type system to be implemented in the class. Thus, the macro accesses the type id and creates the debug symbol by using 'ML_' + the class name. This macro is normally used in implementations of the ML modules.

   If you use

   Example 5.1. `mlDebug`

   ```
   mlDebug("This is the this pointer of this:" << this << ".");
   ```

   in a method or function of your `AddExample` module, the information is printed (provided that the environment variable `ML_AddExample` is not 0 or another debug symbol `ML_AddExample` is defined).

2. `mlDebugConst(ENV_VAR, STREAMOUTS)`

   This macro is used for printing any type of debug information the developer considers to be interesting. The macro scans for the corresponding environment variable `ENV_VAR` or for a debug symbol of the same name registered in the `MLErrorOutput` instance.

   This registering of a debug symbol can also be done in the `CoreControl` module by defining the debug symbol in the "Debug" panel which is the normal way when e.g., using the ML in the application MeVisLab.

   Example 5.2. `mlDebugConst`

   ```
   mlDebugConst("ML_HOST", "Test" << 1 << "Help!");
   ```

   prints "Test1Help!" if the environment variable `ML_HOST` is defined as `!=0` or if a debug symbol of the same name is defined.

3. `mlDebugPrint(STREAMOUTS)`

   This macro is especially designed for ML classes which are not registered in the runtime type system of the ML. It does the same as `mlDebugConst(ML_DEBUG_ENV_NAME, STREAMOUTS)` where `ML_DEBUG_ENV_NAME` must be defined by the programmer before `mlDebugPrint` is called.

   `ML_DEBUG_ENV_NAME` is usually defined once before e.g., a class is implemented. Then `mlDebugPrint(STREAMOUTS)` can be used as long as `ML_DEBUG_ENV_NAME` is undefined. Hence, the programmer does not have to care much about the environment variable for debug outputs and can change it easily without touching any debug print statement.

Example 5.3. `mlDebugPrint`

```
// ... previous code

// Define before class "AddHelper":
#define ML_DEBUG_ENV_NAME "ML_AddHelper"

class AddHelper {
public:
  void testFunction() {
    mlDebugPrint("This is printed if debug symbol ML_AddHelper is defined.");
  }
};

// At end of implementation of AddHelper
#undef ML_DEBUG_ENV_NAME

// next class...
```

To avoid side effects, do not forget to undefine the environment variable at the end of the file.

4. `mlDebugClass(CLASS_NAME, STREAMOUTS)`

   This macro is used to print debug information for a certain class given by `CLASS_NAME`. It requires the runtime type system to be implemented in the class `CLASS_NAME` . Thus the macro accesses the type id and creates the debug symbol from the class name. Hence, symbol-controlled debug outputs for different classes can be mixed.

Example 5.4. `mlDebugClass`

```
//...

mlDebugClass(AddExample, "Debug information for the AddExample class.");

mlDebugClass(AnotherExample, "Debug information for the AnotherExample class.");

///...
```

5. `mlDebugConditional(COND_SYM, STREAMOUTS)`

   This macro is used to specify subsets of debug outputs for a debug symbol given by the runtime type of the class. Debug information is printed if

   - the class name (given by the runtime type) is specified as symbol, or
   - if the class name + "-" + `COND_SYM` is specified.

   If, for instance, the following macro is used in the class `MyModule`:

Example 5.5. `mlDebugConditional`

```
mlDebugConditional("CASES", "Message1");
```

the debug information "Message1" is printed if the debug symbol "ML_MYMODULE" is defined or if the debug symbol "ML_MYMODULE-CASES" is specified. If just "ML_MYMODULE-CASES" is specified, only "Message1" is printed.

This macro requires the runtime type system to be implemented in the class that uses the macro. It accesses the type id and creates the debug environment name.

- `COND_SYM` specifies the additional symbol added to the class symbol (separated by "-").
- `STREAMOUTS` is the stream output sent to the error/debug output if the symbol given by the class name + "-" + `COND_SYM` is activated.

Important

DO NOT implement required functionality in the macro call, because it will not be compiled in release mode.

The code

```
int a=0;
mlDebug("Buggy example, do not use: " << (a=10) << "\n");
int b= a*10;
```

will result in `b == 0` in release mode and in `b == 100` in debug mode.

Note

To make debug outputs more readable, long file names are truncated to 30 characters.

5.2. Handling Errors

The ML provides some functionality for handling errors on different levels. Generally, programmers should not try to handle errors themselves. It is strongly recommended to call the correct handler and leave error handling to the ML.

It is possible to configure the ML so that the application handles errors in different ways: the application could, for instance, generate an e-mail message, or it could terminate, or it could pop up a window and try to continue. The way how applications handle errors should be configured globally for the ML and not for the individual modules.

So, how to handle an error or a warning? There are three macros to be called on warnings, errors or fatal errors:

1. `mlWarning(functionName, errorCode)`

   This macro is used to print warning messages which notify the user or application of any type of (non-urgent) errors or abnormalities. See number 3 for parameter descriptions.

2. `mlError(functionName, errorCode)`

   This macro is used to print errors messages which notify the user or the application of errors that cause incorrect program calculations but that do not lead to program termination, i.e., the system tries to continue processing. However, these errors may lead to fatal errors later. See number 3 for parameter descriptions.

   #### Important

   Do not terminate the program! Leave this decision to the error handling routines of the ML!

3. `mlFatalError(functionName, errorCode)`

   This macro is used to print error messages which notify the user or the application of fatal errors that make it impossible to continue without getting into an invalid program state.

   #### Important

   Do not terminate the program! Leave this decision to the error handling routines of the ML!

   The *`functionName`* string identifies the calling function, including the class name such as "`Host::getTile()`". The *`errorCode`* is an `MLErrorCode` or a string that describes the problem such

as `ML_BAD_DATA_TYPE`, `ML_NO_MEMORY` or `ML_PROGRAMMING_ERROR`. Finally the most important thing is that each of the macros returns a stream object which can be used to provide futher information on the problem. This has the advantage that a complex information string can be streamed into the macro instead of having to create a string by manually.

Note

You are not responsible for the program to continue safely after a fatal error, just explain what you do even if it will lead to a crash.

This is necessary since fatal error management depends on the configuration of the ML error handler. The error handler might try to continue the code normally, or to terminate the program, or to jump into a debugger, to send a mail, to throw an exception or something else. If you terminate the process, the ML will not be able to handle it.

4. `mlInfo(functionName)`

   This macro is used to print any type of non-debugging information to the error handler of the ML. The macro is typically used for important log information that are not warnings or error messages but that are important for application analysis, e.g., after a crash. It returns a stream object which is used to construct the information string.

Important

To avoid the application and MeVisLab log files being filled with useless information during normal operation, do not use this macro for debugging purposes. Use the appropriate `mlDebug` macros instead.

Below you can see some examples that illustrate how to make use of the macros. You can stream any object that supports to be streamed to a std::stream into the result message.

```
mlWarning("SomeObject::someFunction", ML_BAD_PARAMETER) \
  << "The passed parameter " << someIndex << " is out of range.";

mlError("SomeObject::someFunction", ML_NO_MEMORY) \
  << "Could not allocate image of size " << someExtent << ".";

mlInfo("SomeObject::someFunction") << "Finished registration, result matrix is " << someMatrix;
```

5.3. Registering Error Handlers

The ML provides the class `ErrorOutput` and the class `ErrorOutputInfos` for error handling and redirecting ML outputs. It contains a set of methods to print debug information, warnings, errors and fatal errors. There is a registration mechanism where the application can register itself to be notified when an error, a warning or debug information is to be printed or handled.

When you have registered your own error handling function with `ErrorOutput::addErrorOutputCB()`, the class `ErrorOutput` calls this function of type `ErrorOutput::ErrorOutputCB` to notify the application. It passes the registered *`userData`* pointer, a completely composed information string and a structure of type `ErrorOutputInfos` to the function.

A structure of type `ErrorOutputInfos` contains

- a type identifier (warning, error, fatal error or debug),
- a prefix string with arbitrary information printed before a function,
- the function name,
- the error code,

- the reason/info string,
- the string with information about the handling of the error,
- the file name,
- the line number and
- the time stamp when the message was received by the error handler.

See `mlErrorOutputInfos.h` . and `mlErrorOutput.h` in project `MLUtilities` for parameter descriptions.

5.4. The Class `ErrorOutput` and Configuring Message Outputs

The class `ErrorOutput` is the central error handle and redirection class for the ML. It contains a set of methods to print debug and tracing information, warnings, errors and fatal errors, and to configure a lot of error and message output settings, to specify special behavior (e.g., aborting) on some message types and much more. It also offers a registration mechanism where the application can register itself to be notified when an error, tracing, warning or debug information is to be printed or handled. They permit to register functions to notify the application, to control a set of debug environment variables, etc.

The ML provides a global instance `MLErrorOutput` of this class which is normally used by debug and error handling macros as well as by the ML API (see Section 6.3, “mlAPI.h”). You do not have to create an instance on your own.

Note

Usually, you should not use this class directly.

Since this class is subject to change, use debug or error handling macros instead.

If you need to configure the error handling system e.g., for your application, use the `CoreControl` module (if available in your application) or functionality of ML API (see Section 6.3, “mlAPI.h”).

See the header file documentation of `mlErrorOutput.h` and `mlErrorOutputInfos.h` for detailed information on how to use the following methods.

The class `ErrorOutput` offers methods to

- specify what the ML shall do on debug messages, (fatal) errors, etc.

  1. `void setTerminationType(MLMessageType level, MLTerminator term);`
  2. `MLTerminator getTerminationType(MLMessageType level) const;`

- specify a filter to suppress any type of undesired messages (e.g., debug infos, std::cout, std::cerr or other messages)

  1. `void setMessageFilter(unsigned int messageType);`
  2. `unsigned int getMessageFilter();`

- install a dump callback function that is called when any error occurs on a runtime type. It permits dumping a runtime typed object as a string into the error message:

  1. `void setDumpCB(DumpCB *dumpCB);`

2. `DumpCB *getDumpCB() const;`

- enable/disable debug message handling:

1. `void setFullDebuggingOn(bool on);`
2. `bool isFullDebuggingOn() const;`

- manage registered callback functions that will be called when messages are sent to this class:

1. `void addErrorOutputCB(void *userData, ErrorOutputCB *callback);`
2. `void removeErrorOutputCB(void *userData, ErrorOutputCB *callback);`
3. `bool hasErrorOutputCB(void *userData, ErrorOutputCB *callback) const;`
4. `void removeAllErrorOutputCBs();`
5. `unsigned long getNumOutputCBs() const;`

- manage a set of debug symbols (also see Section 5.1, “Printing Debug Information”):

1. `void addDebugEnvName(const std::string &envName);`
2. `void removeDebugEnvName(const std::string &envName);`
3. `bool hasDebugEnvName(const std::string &envName) const;`
4. `const std::vector<std::string> &getDebugEnvNames() const;`
5. `void removeAllDebugEnvNames();`
6. `unsigned long getNumDebugEnvNames() const;`

- suppress std::cout and std::cerr prints to the standard console outputs:

1. `bool areMessagesSentToCout() const;`
2. `void sendMessagesToCout(bool on);`
3. `bool areMessagesSentToCerr() const;`
4. `void sendMessagesToCerr(bool on);`

- send messages to an instance of this class that are used by debug and error handling macros (see also Section 5.2, “Handling Errors” and Section 5.1, “Printing Debug Information”):

1. `void printAndNotify(...) const;`
2. `void handleDebugPrint(...) const;`

- configures and returns which message types shall be dumped:

1. `size_t getTraceDumpMessageBits() const;`
2. `void setTraceDumpMessageBits(MLuint32 bitMask);`

- configure the length of the list of recently called functions and when to dump the current call stack into the registered error handling callbacks (see also TracingAndExceptionHandling):

1. `size_t getMaxNumTraceListDumps() const;`
2. `void setMaxNumTraceListDumps(MLGlobalTraceBufferType num);`

3. `size_t getMaxNumTraceStackDumps() const;`
4. `void setMaxNumTraceStackDumps(MLGlobalTraceBufferType num);`

5.5. Tracing, Exception Handling and Checked Object Construction/Destruction

The ML is a library of base classes that many modules and applications use to implement image processing algorithms. In such a complex system, mechanisms to catch, log and handle runtime errors and crashes as well as mechanisms to trace program execution are required. Especially for critical or potentially unsafe functionality, support for additional checks and controls must be provided. The following paragraph describes some macros that allow for the implementation of highly safe source code with crash and error logging especially for critical functionality:

Tracing Program Execution:

1. `ML_TRACE_IN("<FunctionDescription>")`

   This macro should be implemented as the first line in all functions and methods that are not very time-critical. When this code is compiled in release mode, it implements functionality that pushes a reference to the string `<FunctionDescription>` on a stack and into a list, and when the function is finished the pushed information is popped from the stack. This push/pop/list functionality is implemented in the classes `Trace` and `TraceBuffer` in project `MLUtilities`. The ML error handler (see Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs”) can be configured to append the list of recently called functions (trace list dumps) and the current call stack (trace stack dumps) to the registered error output callbacks for additional bug analysis.

   Note that this macro only uses about 8 simple CPU instructions in release code and thus can be added to most functions without significant performance loss.

2. `ML_TRACE_IN_TIME_CRITICAL("<FunctionDescription>")`

   This macro is identical with the macro `ML_TRACE_IN("")`, however, it is only compiled if explicitly enabled for diagnostic purposes. In normal debug or release mode, this tracing macro is not compiled. It is especially useful for tracing time-critical functionality which is assumed to operate safely in normal mode.

Handling Exceptions:

1. `ML_TRY {`

   This macro opens a source code region to be checked for undesired exceptions. If such an exception occurs, the closing `ML_CATCH*()` macro implements crash handling and error logging with the ML error manager and memory cleanup.

2. `} ML_CATCH()`

   This macro can be used to close an `ML_TRY {` code fragment. The macro sends a fatal error to the ML error manager with `ML_PRINT_FATAL_ERROR()` and continues with the execution of the memory manager which is returned by that macro. It is typically used when no resources that were opened or allocated in the enclosed code need to be cleaned up.

3. `} ML_CATCH_RETURN_NULL()`

   This is another macro that can be used to close an `ML_TRY {` code fragment. It is identical with `} ML_CATCH()`, but it returns 0.

4. `ML_CATCH_BLOCK(<exception type>){ <handling code> }`

This is another macro that can be used to close an `ML_TRY {` code fragment and that allows for cleaning up resources opened or allocated in the enclosed code. Multiple implementations of `ML_CATCH_BLOCK()` can be implemented one after another to handle different types of exceptions.

Note that `ML_CATCH_BLOCK()` **does not post errors** to the ML error manager; this must be done explicitly in the `<handling code>` section if necessary.

Important

The macros listed above implement exception catching and error posting only if the code is compiled in release mode.

In debug mode, the macros result in dummy code which does not perform exception handling or catching, i.e., errors and exceptions will cause normal program crashes. This strategy has been chosen to simplify debugging in debug mode, because detecting precise error positions becomes more difficult in many debugging tools when exception handling is enabled.

The following code fragments demonstrate tracing and exception handling:

Example 5.6. Example of a Typical Use of the ML_TRACE_IN() Without Exception Catching

```
void MyClass::testFunction1()
{
  ML_TRACE_IN("void MyClass::testFunction1()");
  <function body>
}
```

Example 5.7. Example of a Typical Use of the ML_TRACE_IN() with Exception Catching

```
void MyClass::testFunction2()
{
  ML_TRACE_IN("void MyClass::testFunction2()");
  ML_TRY
  {
     <The function body is implemented here. If an exception
      is thrown in it then ML_CATCH posts a fatal error to
      the ML error manager, and - if the error manager does
      not terminate the process - continues execution normally>
  }
  ML_CATCH;        // This catches the error, posts it and continues
                   // if the ML error manager continues execution
}
```

Example 5.8. Example of a Typical Use of the ML_CATCH_RETURN_NULL()

```
int MyClass::testFunction3()
{
  ML_TRACE_IN("int MyClass::testFunction3()");
  ML_TRY
  {
     <The function code is implemented here. If an exception
      is thrown in it then ML_CATCH_RETURN_NULL posts a fatal error to
      the ML error manager, and - if the error manager does
      not terminate the process - continues execution with a
      returning 0>

    return result; // This is the return statement in case of successful execution.
  }
  ML_CATCH_RETURN_NULL;     // This catches the error, posts it and returns
                   // 0 if the ML error manager continues execution
}
```

Note

The semicolons behind the `ML_TRACE_IN()` macros can be omitted but are useful for an automatic code indention by the development environment.

Constructing and Deleting Objects:

1. `ML_CHECK_NEW(ptr, expression)`

   This implements a `new` of the passed *`expression`*. In release mode, it handles the exception with an `ML_PRINT_FATAL_ERROR` post to the ML error manager. The pointer must have been set to `NULL` before.

2. `ML_CHECK_NEW_TH(ptr, expression)`

   This executes a `new` of the passed *`expression`*. In release mode, it handles the exception with a `ML_PRINT_FATAL_ERROR` post to the ML error manager and it throws either an `ML_NO_MEMORY` exception or an `ML_CONSTRUCTOR_EXCEPTION` dependent on whether the `new` statement returned `NULL` or the constructor threw an exception. The pointer must have been set to `NULL` before.

3. `ML_DELETE(ptr)`

   This macro is used to delete an object allocated with `ML_CHECK_NEW(ptr, expression)` or with `ML_CHECK_NEW_TH(ptr, expression)`. It must only be used with a single created object, not with an array (see below).

4. `ML_DELETE_ARRAY(ptr)`

   This macro is used to delete an object allocated with `ML_CHECK_NEW(ptr, expression[<objectNum>])` or with `ML_CHECK_NEW_TH()`. It must only be used for allocated object arrays.

Important

Always try to use the above macros for constructing and deleting objects inside of ML code. In future, this will provide a more powerful and failsafe memory management, and it will also correctly handle and log errors that occur in applications.

Note

See Section 2.2.2, “ `Memory` ” for an alternative memory management concept with ML allocation and freeing statements.

Validating Program States:

1. `ML_CHECK(<expression>)`

   This macro posts an `ML_PRINT_FATAL_ERROR()` to the ML error manager if the passed *`<expression>`* returns `false`. This is the typical way of checking entry conditions in functions, for example.

   If the ML error manager continues execution, normal program execution continues after the `ML_PRINT_FATAL_ERROR()` macro.

2. `ML_CHECK_ONLY_IN_DEBUG(<expression>)`

   This macro is identical with the `ML_CHECK(<expression>)` macro, however, it is only compiled in debug mode. In release mode, it is not implemented at all. So this macro is comparable to the normal `assert()` statement. With the `assert()` statement, however, errors are redirected to `abort()` and not to the ML error manager.

3. `ML_CHECK_THROW(<expression>)`

   This macro posts an `ML_PRINT_FATAL_ERROR()` to the ML error manager if the passed *`<expression>`* returns `false`. This is the typical way of checking program or parameter states in functions for validity.

If the ML error manager continues execution, this macro throws an `ML_BAD_POINTER_OR_0` exception after the `ML_PRINT_FATAL_ERROR()` macro. Thus this macro is especially useful in code segments which are enclosed in `ML_TRY { <function body> } ML_CATCH*()` segments.

Note

Also see Section 5.2, “Handling Errors” for explicit usage of error and warning posts.

Example 5.9. Detailed Example for a Checked Object Allocation with ML_CHECK_NEW_THROW() and Release of Resources on Crashes

```
double MyClass::testFunction4()
{

  int *newArray = NULL;
  double retVal = 0;
  ML_TRY
  {
    // Allocate an integer array with new.
    ML_CHECK_NEW_THROW(newArray, int[200]);

    int result = 0;

    /*
     We assume that the function code makes use of the
     allocated data here and that it must calculate a
     non zero return value; if result remains 0 then
     we have a bug somwhere...
    */

    // This value is expected to be non zero, otherwise
    // we have a fatal error, check it chere.
    ML_CHECK_THROW(result);

    // Calculate the return value.
    retVal = 10. / result;

    // Release the allocated memory and reset pointer.
    ML_DELETE_ARRAY(newArray);
  }
  ML_CATCH_BLOCK(...){

    // Clean up allocated resources after any crash in
    // ML_TRY{ } block if pointer is non NULL.
    ML_DELETE_ARRAY(newArray);

    // Post and log the error.
    ML_CHECK(0);

    // Optionally and dependent on the way how the application
    // handles errors the exception can be propagated to the caller
    // such that it terminates execution until the main function is
    // reached and the program state is cleaned up correctly.
    // Another option would be to continue here.
    throw();
  }

  return retVal;
}
```

Chapter 6. The C-API

Chapter Objectives

By reading this chapter, you will get information on how to use the ML and the ML modules with other languages and without C++.

6.1. The C-API

The ML includes an interface that exports ML functionality as pure C. Many other programming libraries can also use the ML functionality because most linkers can bind pure C objects from different languages if they have a pure C interface. MeVisLab also uses the ML by simply including the files `mlInitSystemML.h` , `mlAPI.h` and `mlDataTypes.h` in pure C mode (see Section 6.2, “mlInitSystemML.h”, Section 6.3, “mlAPI.h” and Section 6.4, “mlDataTypes.h”). The files the files `mlInitSystemML.h` and `mlDatatypes.h`, however, can be both; if setting the compiler switches `ML_DISABLE_CPP`, only the C interface is available; otherwise C++ classes can also be used.

Most of the ML functionality can be accessed by including the three files described in Section 6.2, “mlInitSystemML.h”, Section 6.3, “mlAPI.h” and Section 6.4, “mlDataTypes.h”.

6.2. mlInitSystemML.h

This file provides access to the most basic ML functionality which is system-dependent and defines system-independent settings from it. This includes:

- Import and export symbols for Microsoft® platforms,
- The 64 bit integer data type, some constants and types around it,
- Stream input and stream output for 64 bit integer (C++ mode only),
- All system include files needed by any ML class and most ML modules (C++ mode only), and
- ML initialization and destruction functions in the namespace ml (C++ mode only).

Note

This file is compiled in pure C style if the compiler switch `ML_DISABLE_CPP` is set; if not, it also includes the C++ stuff.

Generally you do not have to care about this file, because the file is included in the correct mode when ML classes are used.

6.3. mlAPI.h

This file provides access to the following ML functionality:

- Initialization and destruction of the ML, module library loading,
- Management of ML modules (creation, deletion, hierarchical/inheritance information of ML modules),
- Accessing parameter fields of ML modules,
- Module persistence: saving/restoring module states,

- Setting/getting values of (parameter) field values, names and types,
- (Dis)connecting and notifying (parameter) fields,
- Requesting, allocating and freeing image data from ML modules,
- Managing (setting limits, clearing, querying) the ML memory cache,
- Requesting image information from output fields of ML modules(extents, image transformations, voxel sizes, DICOM tag list, etc.), and
- Special access to `BaseField`, `DicomTagListField` and `SoNodeField`.

6.4. mlDataTypes.h

This file provides access to the following ML functionality:

- Memory and tile (re)allocation, duplication and freeing,
- ML type system initialization and destruction,
- Querying data type properties: Checks for floating point, integer, standard, sign, size, minimum, maximum, validity, etc. properties,
- Merging types to new types: Promoted precision and types from two other types,
- Information on registered types and their implemented functionality: Bit mask with flags for all implemented operations,
- Functions and macros for the registration of user-defined voxel data types,
- Little/Big endian conversion functionality for data and registered voxel data types,
- Registering a user defined voxel data type, and
- A set of convenience functionality to compute with registered data types, to convert them from/to strings, to allocate/manage/free voxel buffers.

6.5. mlTypeDefs.h

This file contains most definitions, typedefs, structs enums, etc. used by the ML and by `MLUtilities`. This file can be included without having to include or link anything from the ML, `MLUtilities` or `MLLinearAlgebra` (see Section 2.6.2, “ `MLUtilities` ” and Section 2.6.1, “`MLLinearAlgebra`(Vector2, ..., Vector10, Vector16, Matrix2, , ..., Matrix6, quaternion, ImageVector)”) . This permits using ML types without having an actual library dependency.

- Definition of macros, constants, enumerations for ML data types, colors (channels), error codes, etc.,
- Callback types which can be registered in the ML,
- Function (pointer) types for (arithmetic) operations of voxel data types,
- Structure definition describing all type functions, properties and operations, and
- Functions and macros for the registration of user defined voxel data types.

6.6. C-Example using the C-API

The following section contains a small C example that creates an ML module network for loading, filtering and saving an image. Note that the libraries for `MLUtilities`, `MLLinearAlgebra`, `ML`, `MLImageFile`, `MLGeometry1`, `MLDicomTree_OFFIS` and `MLImageIO` must be available in binary search paths to run the program correctly. They can normally be found in the installation directory of MeVisLab which is usually available when working with the ML.

The example program implements the following operations:

- Checking the number of command line parameters and the validity of the ML version,
- Loading the libraries `MLImageFile`, `MLGeometry1`, and `MLDicomTree_OFFIS` to have all required modules linked to the executable,
- Creating an `ImgLoad`, `Resample3D`, and `ImageSave` module,
- Setting input and output file names in `ImageLoad` and `ImageSave` module,
- Connecting `ImageLoad`, `Resample3D`, and `ImageSave` module to a module pipeline, and
- Setting zoom parameter in `Resample3D`, saving the result in a file by triggering `ImageSave`, and checking the status field of `ImageSave`.

Example 6.1. Using the C-API

```
// Simple ML program that initializes the library, loads the given
// dataset, applies a resampling and writes the result back to disk.
//
// The input file can be any format supported by the MFL (MeVis File Library) now called MLImageIO,
// including DICOM (.dcm), TIFF (.tif,.tiff)
//
// The output file is written as a DICOM/TIFF combination typically used by
// MeVisLab (DICOM header + tiff data).

#include "mlAPI.h"

#include <stdio.h>
#include <iostream>
```

```
int main(int argc, char* argv[])
{
  // run only if enough arguments
  if (argc > 5) {

    // Extra char buffer
    char  buffer[4096]="\n";

    std::cout << "imagefilter: loading " << argv[1] << std::endl;
    std::cout << "imagefilter: output  " << argv[2] << std::endl;

    // Initialize the ML.
    MLInit(ML_MAJOR_VERSION, ML_MAJOR_CAPI_VERSION, ML_CAPI_REVISION);

    // Load additional image file and filter module libraries.
    MLLoadLibrary("MLImageFile");
    MLLoadLibrary("MLGeometry1");
    // Also load a DICOM tree implementation to be able to load DICOM images.
    MLLoadLibrary("MLDicomTree_OFFIS");

    //------------------------------------------
    // Create modules

    // Create an ImgLoad module.
    mlModule*  loader   = MLCreateModuleFromName("ImgLoad");
    // Create resample module.
    mlModule*  resample = MLCreateModuleFromName("Resample3D");
    // Create an ImgSave module.
```

```
    mlModule*  writer   = MLCreateModuleFromName("ImgSave");

    //-----------------------------------------
    // Setup file names

    // Get the file name field of the loader.
    mlField* loaderFilenameField = MLModuleGetField(loader,"filename");
    // Set the file name field to the given command line argument
    MLFieldSetValue(loaderFilenameField,argv[1]);

    // Get the file name field of the writer.
    mlField* writerFilenameField = MLModuleGetField(writer,"filename");
    // Set the file name field to the given command line argument
    MLFieldSetValue(writerFilenameField,argv[2]);

    //-----------------------------------------
    // Connect modules

    // Get the output image field of the loader.
    mlField* loaderOutput0  = MLModuleGetField(loader,"output0");
    // Get the input image field of the resample module.
    mlField* resampleInput0   = MLModuleGetField(resample,"input0");
    // Connect input of resample to output of loader.
    // Always connect input to output (destination to source) and not vice versa.
    MLFieldConnectFrom(resampleInput0,loaderOutput0);

    // Get the output image field of the resample module.
    mlField* resampleOutput0  = MLModuleGetField(resample,"output0");
    // Get the input image field of the writer module.
    mlField* writerInput0   = MLModuleGetField(writer,"input0");
    // Connect input of resample to output of loader.
    MLFieldConnectFrom(writerInput0,resampleOutput0);

    //-----------------------------------------
    // Set zoom factor

    // Get zoom factor field.
    mlField* zoomField = MLModuleGetField(resample,"zoom");

    // Concatenate arguments to form a vector string.
    sprintf(buffer,"%s %s %s",argv[3],argv[4],argv[5]);

    // Set vector string value to zoom field.
    MLFieldSetValue(zoomField,buffer);

    //-----------------------------------------
    // Write image back to disk

    // Get save field from writer.
    mlField* saveField = MLModuleGetField(writer,"save");
    // Touch the save trigger, this actually saves the image to disk.
    std::cerr << "Starting image save..." << std::endl;
    MLFieldTouch(saveField);
    std::cerr << "...finished." << std::endl;

    //-----------------------------------------
    // Check if writing was ok
    mlField* statusField = MLModuleGetField(writer,"status");
    // Get value of status field into given buffer (maximum buffer size is also passed).
    MLFieldGetValue(statusField, buffer, 4096);

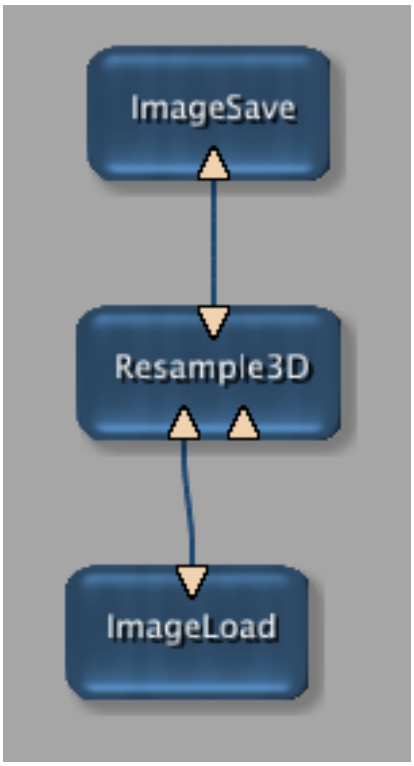
    std::cout << "Write status: " << buffer << std::endl;

  } else {
    std::cout << "Usage: imagefilter inputfile outputfile xscale yscale zscale" << std::endl;
  }
  return 0;
}
```

This example called with the command line arguments

```
/demodata/Carotid1_MRA.small.dcm   Carotid1_MRA.small.scaled.dcm  1 2 3
```

is comparable to the following module network and panels in MeVisLab:


ImageSave
Resample3D
ImageLoad

Chapter 7. Registered Voxel Data Types

Chapter Objectives

This chapter gives an introduction to programming, implementing and registering user-defined data types for voxels:

- Section 7.1, “Overview of Registered Voxel Data Types”
- Section 7.2, “Implementing Image Processing on extended Voxel Data Types”
- Section 7.3, “Limitations of Registered Data Types”
- Section 7.4, “Traps and Pitfalls When Using Registered Voxel Types”
- Section 7.5, “Advanced Issues on Registered Voxel Types”

See Section 7.2.3, “Examples with Registered Voxel Types” for code examples.

7.1. Overview of Registered Voxel Data Types

ML modules normally implement algorithms on integer or floating point typed voxels, such as `MLint8`, `MLuint16` or `MLfloat`. To support all these types, the image processing parts of ML modules algorithms normally use templates. Modules can also support other, extended data types like Vector3 or Matrix3, but it is not very efficient to use the template mechanism if a module is to support any extended voxel type. In this case a module should not use the types directly but rather use the type operations table that is registered for every type supported by the ML.

Using these tables is somewhat cumbersome, but is the only way to support types that are not even registered yet.

This means:

- The number of registered voxel types is potentially unlimited.
- A programmer can add his own voxel types.
- An image processing algorithm can also use explicit voxel types without use the type operation tables which is less universal, but usually faster, because the compiler can do more optimizations.
- Operations that are defined on the SubImage class make use of the registered types.

Application areas for new voxel types could be vector field processing, color voxel filters, voxels with segmentation information (like bit fields, object indices, etc.), matrix/tensor images, complex numbers, quaternions, strings as voxels and many more.

7.1.1. Registered Voxel Data Types

The following voxel types are already registered in the ML:

1. - `complexf`, and
   - `complexd`.

   Complex numbers use float and double floating point arithmetics. They make the standard C++ complex data type available and are implemented in the ML as a template class `MLTComplexTypeInfos` in project `MLTypeExtensions`.

2. - `quaternionf`, and
   - `quaterniond`.

   Quaternions use float and double floating point arithmetics. They make the quaternion data type (from project `MLLinearAlgebra`) available and are implemented in the ML as a template class `MLTQuaternionTypeInfos` in project `MLTypeExtensions`.

3. - `vecf2` and `vec2`,
   - `vecf3` and `vec3`,
   - `vecf4` and `vec4`,
   - `vecf5` and `vec5`,
   - `vecf6` and `vec6`,
   - `vecf7` and `vec7`,

- `vecf8` and `vec8`,
- `vecf9` and `vec9`,
- `vecf10` and `vec10`,
- `vecf16` and `vec16`,
- `vecf32` and `vec32`, and
- `vecf64` and `vec64`.

These voxel types as well as some other specializations of the `ScalarVectorTemplate` (from project `MLLinearAlgebra`) for higher vector dimensions and for float and double data types. They are implemented in the ML as specializations of the template class `MLTDoubleVectorTypeInfos` in project `MLTypeExtensions`.

4.
   - `matf2` and `mat2`,
   - `matf3` and `mat3`,
   - `matf4` and `mat4`,
   - `matf5` and `mat5`, and
   - `matf6` and `mat6`.

   These matrix voxel types are implemented in the ML as a template class `MLTMatrixTypeInfos` in project `MLTypeExtensions`. The used base types can be found in the project `MLLinearAlgebra`..

For the registration of these classes, the class `TypeInfosBase` has been implemented in the project `MLTypeExtensions`.

Note

The standard data types `MLuint8`, `MLint8`, `MLuint16`, `MLint16`, `MLuint32`, `MLint32`, `MLint64`, `MLfloat` and `MLdouble` are also registered for the sake of completeness. Thus it is possible to request their type properties as with all the other registered data types.

The type information for the standard types are implemented in the ML as specializations of the template class `MLTStdTypeInfos`.

7.1.2. About Standard, Default and Registered Voxel Types

There are different voxel types sets in the ML.

- **Scalar Voxel Types**

  Standard voxel types are the "normal" data types. They are available in many programming languages, such as signed and unsigned 8, 16, 32 and/or 64 bit sized integers, float and double types. They are also the most typical types used for image voxels.

- **Default Voxel Types**

  The default voxel types contains besides the scalar voxel types also some selected extended voxel types. These are `std::complex<float>`, `std::complex<double>`, `Vector2f`, `Vector2d`, `Vector3f`, `Vector3d`, `Vector6f`, `Vector6d`, `Matrix2f`, `Matrix2d`, `Matrix3f` and `Matrix3d`. This should cover the most common uses e.g. for tensor imaging or complex typed voxels.

- **Registered Voxel Types**

Registered voxel types are loaded to the application code at runtime. Each registered type provides a function table with basic functions for data addition, subtraction, multiplication, shift and so on.

See Section 7.5.1, “About the Difference Between Scalar, Extended and Registered Voxel Types” for a detailed voxel type discussion.

7.2. Implementing Image Processing on extended Voxel Data Types

This section gives detailed information on programming with extended voxel types.

This includes

- configuring your module to work fine with extended voxel types,
- handling compile and runtime decisions between scalar and extended voxel types and their properties,
- getting and managing metadata on extended voxel types,
- working with templates on extended voxel types outside the template function `calculateOutputSubImage`,
- handling generalized registered voxel types and module parameters with `DataTypeField` and `UniversalTypeField`, and
- advanced configuration and programming issues.

See Section 7.2.3, “Examples with Registered Voxel Types” for examples.

7.2.1. Important Functions For Voxel Types

The ML provides many helpful functions that support managing different voxel types and using them for programming (see Section 7.5.2, “Getting and Managing Metadata About Registered Voxel Types” for a detailed discussion).

The most important functions are:

- `size_t MLSizeOf(MLDataType dt)`

  returns the size of the data type *`dt`* in bytes. On invalid types 0 is returned.

- `MLDataType MLGetDataTypeFromName("data_type_name")`

  determines the data type id of the type to be handled, because it is not available as a precompiled constant.

- `bool MLIsValidDataType(MLDataType dt)`

  checks whether the data type is registered.

- `bool MLIsStandardType(MLDataType dt)`

  checks whether the data type *`dt`* is a normal built-in compiler type.

- `MLTypeInfos* MLGetTypeInfosForDataType(MLDataType outDType)`

  returns a pointer to the `MLTypeInfos` object, which describes features and properties of the data type, or returns NULL if *`outDType`* is an undefined data type.

The following methods of the class `PagedImage` are normally used in the `calculateOutputImageProperties` method of self-developed ML module classes when data types are not appropriate for the implemented algorithm:

- `PagedImage::setInvalid()`

  invalidates the module output if the module cannot operate, because e.g., the type does not exist or the data type is not appropriate for the algorithm.

- `PagedImage::setStateInfo(<message>, ML_TYPE_NOT_REGISTERED)`

  specifies the reason in *`<message>`* why the output image has been invalidated. A connected `Info` module, for example, will show the reason in its state information.

See Section 7.5.2, “Getting and Managing Metadata About Registered Voxel Types” for information on further functions.

7.2.2. The Basic Concept of Calculating the Output SubImage

After the output properties were evaluated in `calculateOutputImageProperties` the output image will be requested by calling the derived function `calculateOutputSubImage(SubImage *outSubImg, int outIndex, SubImage *inSubImgs)` of the module. This function was generated in MeVisLab before versions 3.6 by a set of preprocessor macros, e.g. `ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_SCALAR_TYPES_CPP`. Those macros ensured that for all possible input and output voxel types the function temmplate `calculateOutputSubImage` gets instantiated and called during runtime. These macros and not necessary anymore with the possibilities of C++17 but are kept for backward compatibility. The file `mlTSubImageVariant.h` contains a set of functions. The description of each function contains an example of its usage. These functions can create from a SubImage a variant kind type that can be passed to `std::visit` and that does the necessary dispatching to the different instantiations of `calculateOutputSubImage`

7.2.3. Examples with Registered Voxel Types

The following examples contain many useful code fragments for handling and using registered voxel types. For advanced examples see

- Section 7.2.5, “Handling Generalized Registered Voxel Types as Module Parameters” to implement general module fields that handle parameters for any registered or any standard voxel type.
- Section 7.5.3, “Reducing Generated Code and Compile Times” to have module implementations compile only standard, only registered or only self-defined voxel types and by that to reduce both compile time and size of the generated code.
- Section 7.4, “Traps and Pitfalls When Using Registered Voxel Types” to see the difference between correct and incorrect pointer incrementation when traversing and accessing registered voxels in a templated `calculateOutputSubImage` function.
- Section 7.4, “Traps and Pitfalls When Using Registered Voxel Types” to implement and register your own voxel type.

Example 7.1. How to Check and Set a Registered Type Safely as the Output Voxel Type in calculateOutputImageProperties

```
MLDataType dt = MLDataTypeFromName("vecf3");
if (!MLIsValidType(dt)){
  outImage->setInvalid();
  outImage->setStateInfo("Could not find type 'vecf3'", ML_TYPE_NOT_REGISTERED);
  return;
}
outImage->setDataType(dt);
```

This example shows how to select a specific voxel type for the output image. Note that a registered voxel type is used whose id is unknown at compilation time. That is why the voxel type id is determined by using the function `MLDataTypeFromName`.

Example 7.2. How to Write calculateOutputSubImage without Macros

```
void SetVoxelValue::calculateOutputSubImage(SubImage *outSubImg, int outIndex, SubImage *inSubImgs)
{
  auto imagePair =
      createTSubImageVariantPair<MLuint8, MLint8, MLuint16, MLint16, MLuint32, MLint32, MLuint64,
                                 MLint64, std::complex<MLfloat>, std::complex<double>, Vector2f,
                                 Vector2d, Vector3f, Vector3d, Vector6f, Vector6d, Matrix2f,
                                 Matrix2d, Matrix3f, Matrix3d>(*outSubImg, inSubImgs);

  auto visitor = [this, outIndex](auto& ip){ calculateOutputSubImage(ip.output, outIndex, ip.input); };

  std::visit(visitor, imagePair);
}
```

The template parameters of the function `createTSubImageVariantPair` specifies all possible voxel types that this module can support.

```
template <class DATATYPE>
void SetVoxelValue::calculateOutputSubImage(TSubImage<DATATYPE>& outImg, int /*outIdx*/, const TSubImage<DATATYPE>&  /*inImg*
{
  if (outImg.getBox().contains(_inputVoxelPos))
  {
    outImg.setImageValue(_inputVoxelPos, *(reinterpret_cast<DATATYPE*>(_writeValueFld->getUniversalTypeValue())));
  }
}
```

Example 7.3. How to Write calculateOutputSubImage for Different Input and Ouput Voxel Types Without Macros

```
void DifferentTypesInputOutputExample::calculateOutputSubImage(SubImage *outSubImage, int outIndex,
                                                               SubImage *inSubImage)
{
  auto input =
      createTSubImageVariant<MLuint8, MLint8, MLuint16, MLint16, MLuint32, MLint32, MLuint64, MLint64,
                      MLfloat, MLdouble>(inSubImage);
  auto output =
      createTSubImageVariant<MLuint8, MLint8, MLuint16, MLint16, MLuint32, MLint32, MLuint64, MLint64,
                      MLfloat, MLdouble>(outSubImage);

  auto visitor = [this, outIndex](auto& out, const auto& in){ calculateOutputSubImage(out, outIndex, in); };

  std::visit(visitor, output, input);
}
```

The C++ function `std::visit` creates the cross product of all possible input- and output types.

```
template <typename T, typename U>
void DifferentTypesInputOutputExample::calculateOutputSubImage(TSubImage<T>& outputSubImage,
                                                               int outputIndex,
                                                               const TSubImage<U>& inputSubImage)
{
  const T constantValue = static_cast<T>(_constantValueFld->getDoubleValue());

  // Clamp box of output image against image extent to avoid that unused areas are processed.
  const SubImageBox validOutBox = outputSubImage.getValidRegion();

  // Process all voxels of the valid region of the output page.
  ImageVector p;
  for (p.u = validOutBox.v1.u; p.u <= validOutBox.v2.u; ++p.u)
  {
    for (p.t = validOutBox.v1.t; p.t <= validOutBox.v2.t; ++p.t)
    {
      for (p.c = validOutBox.v1.c; p.c <= validOutBox.v2.c; ++p.c)
      {
        for (p.z = validOutBox.v1.z; p.z <= validOutBox.v2.z; ++p.z)
        {
          for (p.y = validOutBox.v1.y; p.y <= validOutBox.v2.y; ++p.y)
          {

            p.x = validOutBox.v1.x;
            // Get pointers to row starts of input and output sub-images.
            const U *inVoxel0 = inputSubImage.getImagePointer(p);

            T *outVoxel = outputSubImage.getImagePointer(p);

            const MLint rowEnd = validOutBox.v2.x;

            // Process all row voxels.
            for (; p.x <= rowEnd; ++p.x, ++outVoxel, ++inVoxel0)
            {
              *outVoxel = *inVoxel0 + constantValue;
            }
          }
        }
      }
    }
  }
}
```

Example 7.4. How to Accept Non-Standard Input Voxels Only

```
  MLDataType dt = getInputImage(0)->getDataType();
  if (MLIsValidType(dt) && !MLIsStandardType(dt)){
    outImage->setDataType(dt);
  }
  else{
    // Invalidate output image if we have an invalid or a standard voxel data type.
    outImage->setInvalid();
    outImage->setStateInfo("Bad input voxel type", ML_BAD_PARAMETER);
  }
```

This is a similar example which demonstrates how to configure an ML module to accept only registered voxel types in the input image. (The "!" before `MLIsStandardType()` can be removed in order to have the ML module accept only standard types).

Example 7.5. How to Implement a Flip of a Vector3f in calculateOutputSubImage

```
template <typename DTYPE>
  void Vecf3Flip::calculateOutputSubImage(TSubImage<DTYPE> *outSubImg,
                                  int outIndex,
                                  TSubImage<DTYPE> *inSubImg1)
{
  // NOTE: In this example we assume that we have set to operate only on Vector3f voxels.

  // Clamp our page region to the image extent to avoid processing of regions outside the image.
  SubImageBox outBox = outSubImg->getValidRegion();

  // Iterate over all voxels of the valid area of the output subimage.
  ImageVector p;
  for (p.u=outBox.v1.u; p.u<=outBox.v2.u; ++p.u) {
    for (p.t=outBox.v1.t; p.t<=outBox.v2.t; ++p.t) {
      for (p.c=outBox.v1.c; p.c<=outBox.v2.c; ++p.c) {
        for (p.z=outBox.v1.z; p.z<=outBox.v2.z; ++p.z) {
          for (p.y=outBox.v1.y; p.y<=outBox.v2.y; ++p.y) {

            // Get start position of voxel rows in input and in output image.
            p.x = outBox.v1.x;
            DTYPE *iVoxel = inSubImg1->getImagePointer(p);
            DTYPE *oVoxel = outSubImg->getImagePointer(p);

            // Flip all voxels in the row.
            // Warning: Do not iterate with vef3 pointers, because they might have
            // smaller size than DTYPE.
            for (;    p.x <= outBox.v2.x;    ++p.x, ++iVoxel, ++oVoxel ) {

              // Flip Vector3f components from input to output.
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[0] = ( *reinterpret_cast<Vector3f*>(iVoxel) )[2];
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[1] = ( *reinterpret_cast<Vector3f*>(iVoxel) )[1];
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[2] = ( *reinterpret_cast<Vector3f*>(iVoxel) )[0];
            }
          }
        }
      }
    }
  }
}
```

This example shows a possible way of how to implement the template function `calculateOutputSubImage` to flip the three components of a `Vector3f`.

Example 7.6. How to Request a Specific Voxel Type

```
void ExampleModule::calculateOutputImageProperties(int outIndex, PagedImage* outImg)
{
  // Force the input voxel data type to be of Vector2f; set it for image at index 0
  // because we have only one input image.

  outImg->setInputSubImageDataType(0, MLDataTypeFromName("vecf2"));

}
```

This example demonstrates how to implement `calculateOutputImageProperties` to request a specific voxel type for the input subimage. If the input type does not match the requested type, the ML will automatically cast the voxels. Normally, this is done component-wise for registered voxel types. Be aware of the following:

- The algorithm must use the `ML_CALCULATE_OUTPUTSUBIMAGE_NUM_INPUTS_*_DIFFERENT_INPUT_DATATYPES` macros to be able to handle different types of input and output subimages.

- The template function `calculateOutputSubImage` must use two template parameters to distinguish the two types. See documentation of the `ML_CALCULATE_OUTPUTSUBIMAGE_NUM_INPUTS_*_DIFFERENT_INOUT_DATATYPES` macros.
- Some compilers have problems with the large amount of generated code. See Traps And Pitfalls When Using Registered Voxel Types for solutions.
- Implicit casts between registered voxel type are relatively slow, because they are done component-wise.

Example 7.7. How to Convert a vecf2 to a vecf3

```
// Use a macro to call the template function with different input and output template arguments.
ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_DIFFERENT_DEFAULT_INOUT_DATATYPES_CPP(Vecf2ToVecf3Converter);

template <typename OTYPE, typename ITYPE>
  void Vecf2ToVecf3Converter::calculateOutputSubImage(TSubImage<DTYPE> *outSubImg,
                                                      int outIndex,
                                                      TSubImage<DTYPE> *inSubImg1)
{
  // NOTE: In this example we assume that we have set Vector2f as input and Vector3f as output type.

  // Clamp our page region to the image extent to avoid processing of regions outside the image.
  SubImageBox outBox = outSubImg->getValidRegion();

  // Iterate over all voxels of the valid area of the output subimage.
  ImageVector p;
  for (p.u=outBox.v1.u; p.u<=outBox.v2.u; ++p.u) {
    for (p.t=outBox.v1.t; p.t<=outBox.v2.t; ++p.t) {
      for (p.c=outBox.v1.c; p.c<=outBox.v2.c; ++p.c) {
        for (p.z=outBox.v1.z; p.z<=outBox.v2.z; ++p.z) {
          for (p.y=outBox.v1.y; p.y<=outBox.v2.y; ++p.y) {

            // Get start position of voxel rows in input and in output image.
            p.x = outBox.v1.x;
            ITYPE *iVoxel = inSubImg1->getImagePointer(p);
            OTYPE *oVoxel = outSubImg->getImagePointer(p);
            for (;    p.x <= outBox.v2.x;    ++p.x, ++iVoxel, ++oVoxel ) {

              ( *reinterpret_cast<Vector3f*>(oVoxel) )[0] = ( *reinterpret_cast<Vector2f*>(iVoxel) )[0];
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[1] = ( *reinterpret_cast<Vector2f*>(iVoxel) )[1];
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[2] = ( *reinterpret_cast<Vector2f*>(iVoxel) )[0] *
                                                            ( *reinterpret_cast<Vector2f*>(iVoxel) )[1];
            }
          }
        }
      }
    }
  }
}
```

This example shows how to implement `calculateOutputSubImage` to convert a `vec2f` to a `vec3f` by writing the product of the first two vector components into the third one. Note that this example still compiles all possible combinations of input and output voxel types, although only one specific combination is used. This version might be useful when other algorithm parts still use other type combinations, otherwise the following version is recommended.

Example 7.8. How to Convert a Vector2f to a Vector3f Without Template Code

```
void Vecf2ToVecf3Converter::calculateOutputSubImage(SubImage *outSubImg, int outIndex, SubImage *inSubImgs)
{
  // NOTE: In this example we assume that we have set Vector2f as input and Vector3f as output type.

  // Clamp our page region to the image extent to avoid processing of regions outside the image.
  SubImageBox outBox = outSubImg->getValidRegion();

  // Get the sizes of the input and output voxels.
  const size_t iVoxSize = MLSizeOf(inSubImgs->getDataType());
  const size_t oVoxSize = MLSizeOf(outSubImg->getDataType());

  // Iterate over all voxels of the valid area of the output subimage.
  ImageVector p;
  for (p.u=outBox.v1.u; p.u<=outBox.v2.u; ++p.u) {
    for (p.t=outBox.v1.t; p.t<=outBox.v2.t; ++p.t) {
      for (p.c=outBox.v1.c; p.c<=outBox.v2.c; ++p.c) {
        for (p.z=outBox.v1.z; p.z<=outBox.v2.z; ++p.z) {
          for (p.y=outBox.v1.y; p.y<=outBox.v2.y; ++p.y) {

            // Get start position of voxel rows in input and in output image.
            p.x = outBox.v1.x;
            MLTypeData *iVoxel = static_cast<MLTypeData*>(inSubImgs->getImagePointer(p));
            MLTypeData *oVoxel = static_cast<MLTypeData*>(outSubImg->getImagePointer(p));
            for (;  p.x <= outBox.v2.x;  ++p.x) {

              ( *reinterpret_cast<Vector3f*>(oVoxel) )[0] = ( *reinterpret_cast<Vector2f*>(iVoxel) )[0];
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[1] = ( *reinterpret_cast<Vector2f*>(iVoxel) )[1];
              ( *reinterpret_cast<Vector3f*>(oVoxel) )[2] = ( *reinterpret_cast<Vector2f*>(iVoxel) )[0] *
                                                            ( *reinterpret_cast<Vector2f*>(iVoxel) )[1];
              iVoxel += iVoxSize;
              oVoxel += oVoxSize;
            }
          }
        }
      }
    }
  }
}
```

This example explicitly implements the virtual method `calculateOutputSubImage` without using any `ML_CALCULATE_OUTPUTSUBIMAGE` macro. Note that we do not have explicit voxel types anymore. We must use the untyped (void) versions to get voxel positions to the raw data and the sizes of the voxels to move pointers correctly. However, the amount of generated code is considerably smaller, and the compile times are faster.

7.2.4. Compile and Runtime Decisions on Standard and Registered Voxel Types

In order to optimize an algorithm, either with regard to performance or with regard to precision, it is sometimes useful to distinguish between data types or between data type properties. A typical example is: the programmer would like to know whether the template type is an integer, a floating point, a registered, a signed or an unsigned type.

The ML provides a number of functions that return flags depending only on the pointer type; the pointer value is ignored:

- `MLIsStandardTypePtr (const T* ptr),`
- `MLIsSignedTypePtr (const T* ptr),`
- `MLIs8_16_Or_32BitIntegerTypePtr (const T* ptr),`
- `MLIs8BitIntegerTypePtr (const T* ptr),`
- `MLIs16BitIntegerTypePtr (const T* ptr),`
- `MLIs32BitIntegerTypePtr (const T* ptr),`
- `MLIs64BitIntegerTypePtr (const T* ptr),`
- `MLIsBuiltInIntegerTypePtr (const T* ptr),`
- `MLIsBuiltInFloatingPointTypePtr (const T* ptr),`

The following functions return other values such as data type enumerators and sizes, or they activate function tables for registered types:

- `MLGetDataTypeFromPtr (const T* ptr),`
- `MLGetDataTypeSizeFromPtr (const T* ptr),`

Note

The above functions are traits, i.e., they are constant at compile time and can be "optimized away" by compilers. Hence, these functions can even be used in time-critical code.

7.2.5. Handling Generalized Registered Voxel Types as Module Parameters

Some modules require an arbitrary voxel type and its values to be selected and handled. The ML offers the fields `MLDataTypeField` and `UniversalTypeField` to meet this requirement.

1. An `EnumField` can simply be configured to offer a selectable list of all standard and registered voxel types to the user.

2. A `UniversalTypeField` allows to handle a value from a freely selectable `MLDataType`; it also - with certain limitations - implicitly converts data from one type to another when its data type is changed. The filling of values in arbitrarily typed images, for example, can easily be specified, even for registered voxel types.

3. An `MLDataTypeField` stores an `MLDataType` value; it is useful whenever any data type needs to be specified, for example for output images and internal buffers. It is rarely used because in most cases the first version with an `EnumField` version is safer and easier for module users, because there is no need to write the string name of the type correctly.

The following code fragments show how to configure the output image of an ML module with one output and with a fill value of an arbitrary standard or registered voxel type.

Header file:

```
//! Field containing the type of the selected voxel type. Default is MLdoubleType.
EnumField *_voxTypeFld;

//! Field containing the type of the selected voxel type. Default is 0.
UniversalTypeField *_voxValFld;
```

C++-File, `Constructor`:

```
handleNotificationOff();

// Add voxel type field by using the string table of all standard and registered voxel
// types and its size. Also set the default to the double voxel type.
_voxTypeFld = addEnum("voxelType", MLDataTypeNames(), MLNumDataTypes());
_voxTypeFld->setEnumValue(MLdoubleType);
_voxTypeFld->attachField(getOutputImageField(0));

// Add a field to the module which contains a value of the selected data type.
_voxValueFld = addUniversalType("voxelValue");
_voxValueFld->setDataType((MLDataType)(_dataTypeFld->getEnumValue()));
_voxValueFld->setStringValue("0");
_voxValueFld->attachField(getOutputImageField(0));

handleNotificationOn();
```

C++-File, `handleNotification`:

```
// Be sure that the UniversalType field is always of the selected voxel type.
if (field == _voxTypeFld){
  _voxValueFld->setDataType((MLDataType)(_voxTypeFld->getEnumValue()));
}

if (field == _voxValueFld){
  // Get the value of the selected data type as string.
  std::string strVal = _voxValueFld->getStringValue();

  // Get a pointer to memory containing the value of the selected type.
  MLTypeData *fillVal = _voxValueFld->getUniversalTypeValue();
}
```

C++-File, `calculateOutputImageProperties`:

```
// Set output image to the selected data type.
outImg->setDataType ((MLDataType)(getDataTypeFld()->getEnumValue()));
```

C++-File, `calcOutSubmage`:

```
// Fill output subimage with the user defined value.
```

```
  outSubImg->fill(*((DATATYPE*)(getFillExtValueFld()->getUniversalTypeValue())));
```

7.3. Limitations of Registered Data Types

Registered voxel types have some limitations:

- Since modules do not inline the code of registered voxel data types, one voxel operation requires one call to the registered operation, i.e., registered data types are slightly slower than built-in data types. This is usually is not a problem because these operations are often complex so that the call itself is not that expensive compared to the real voxel operation. To achieve maximum performance, a module can also implement specialized code that does not work via registered types.
- For performance and technical reasons, instances of new voxel data types must have a constant size, they cannot have dynamic members, and `memcpy`() must be able to copy them without using copy constructors. (So-called "POD types", i.e. "plain old data" types.)

7.4. Traps and Pitfalls When Using Registered Voxel Types

You might experience some problems when using or implementing modules with registered voxel types. The following hints might help you to solve these problems:

1. There are no extended voxel types available, even modules like `ConstantImage` or `ImagePropertyConvert` do not offer them in their data type selection.

   - Check whether the loading of extended voxel types has been suppressed.
   - Check whether the library `MLTypeExtensions` is available in the search paths of *MeVisLab*. It contains the code for the registered voxel types generally used.
   - Check whether the application or *MeVisLab* loads `MLTypeExtensions` before the types are used. In the case of `MeVisLab`, the corresponding `.def` file must specify the tag `PreloadDLL` to force loading at application startup.

2. The compiler fails "complaining" that the generated code is too large or too complex.

   - Template functions must often be instantiated for all types or even for all combinations of two template types. This can lead to a significant amount of code which exceeds the predefined limits of the compiler. Check the following options:

     a. Simplify the generated code or template function. This can either be done by simplifying the code itself, or by moving code into non-templated functions, if possible, or by reducing compilation to the really needed types. See Section 7.5.3, “Reducing Generated Code and Compile Times” for more information.

     b. Increase the compiler limits. This can be done in MeVisLab projects, for example, by setting `MSVC_COMPILERSTACK = 800` or higher before the file includes in the `.pro` file.

7.5. Advanced Issues on Registered Voxel Types

The following paragraphs describe some features for advanced configurations of your ML module. This includes:

- a detailed description of the differences between standard, extended and registered voxel types (Section 7.5.1, “About the Difference Between Scalar, Extended and Registered Voxel Types”),
- information on how to get and manage metadata about registered voxel types (Section 7.5.2, “Getting and Managing Metadata About Registered Voxel Types”),
- information on how to reduce generated code and shorten compile times (Section 7.5.3, “Reducing Generated Code and Compile Times”),
- information on how to configure supported voxel types (Section 7.5.4, “Configuration of Supported Voxel Types”),
- information on how to implement a new voxel data type(Section 7.5.5, “Implementing a New Voxel Data Type by Deriving from MLTypeInfos”).

7.5.1. About the Difference Between Scalar, Extended and Registered Voxel Types

There are three different kinds of voxels types you need to distinguish when you want to understand how the ML works in detail.

- **Scalar Voxel Types**

  Scalar voxel types are primitive data types. They are available in many programming languages, such as signed and unsigned 8, 16, 32 and/or 64 bit sized integers, float and double types. They are also the most typical types used for image voxels.

  In the ML, these types are called `MLuint8`, `MLint8`, `MLuint16`, `MLint16`, `MLuint32`, `MLint32`, `MLint64`, `MLfloat`, and `MLdouble`. There are also enumerator constants called `MLuint8Type`, `MLint8Type`, `MLuint16Type`, `MLint16Type`, `MLuint32Type`, `MLint32Type`, `MLfloatType`, and `MLdoubleType`, respectively.

  Note

  For compatibility reasons, the `MLuint64` type is not supported in the ML.

- **Extended Voxel Types**

  Extended voxel types are all types that are composed of more than one component, e.g. complex, quaternion, vector or matrix types.

  There is a set of default extended types that is supported by some macros that are used to instantiate template methods for image calculation: `std::complex<float>`, `std::complex<double>`, `Vector2f`, `Vector2d`, `Vector3f`, `Vector3d`, `Vector6f`, `Vector6d`, `Matrix2f`, `Matrix2d`, `Matrix3f` and `Matrix3d`. Apart from these macro where these types are 'hardcoded', these types have no other special meaning.

- **Registered Voxel Types**

  Registered voxel types are loaded to the application code on runtime. Each registered type provides a function table with functions for data addition, subtraction, multiplication, shift and so on. This table

can be used to perform operations on this type.They also provide an `MLTypeInfo` data structure describing their properties, such as name, number of components, size, etc.

The pre-registered types all have enumerators, type traits descriptions (via the `TypeTraits` template class) and type names that can be used in code directly.

7.5.2. Getting and Managing Metadata About Registered Voxel Types

The ML provides a number of functions to analyze, convert, process and manage a data type and its values as well as the components of these values.

These functions are useful for building modules that apply abstract operations on arbitrary data types, for example decomposing a voxel of any data type into its components or casting any arbitrary registered data type to another one.

Note

All these functions are part of the C-API of the ML. Hence they can also be used for managing voxel data in C programs or in modules that do not include the C++ API of the ML.

You do not have to distinguish between scalar and registered voxel data. The following functionality also works fine on scalar voxel types and data.

The most important functions

- `MLSizeOf (MLDataType dt)`,
- `MLGetDataTypeFromName (const char* dtName)`,
- `MLIsValidDataType (MLDataType dt)`,
- `MLIsScalarType (MLDataType dt)`,
- `MLGetTypeInfosForDataType (MLDataType dt)`, and

which are often used for module development, are described in Section 7.2.1, “Important Functions For Voxel Types”.

7.5.2.1. Functions for Managing Components of Registered Voxel Types

Functions for managing voxel components:

- `const char* MLGetCDataTypeNameForCharCode(char code);`

  Returns the basic C/C++ data type name corresponding to a character representing it. On invalid codes "" is returned.

- `const char* MLGetMLDataTypeNameForCharCode(char code);`

  Returns an ML type name compatible with a character representing it. On invalid codes "" is returned. The return value match for function calls to `MLDataTypeFromName()`.

- `MLDataType MLGetMLDataTypeForCharCode(char code);`

  Returns an ML data type compatible with a character representing it. On invalid codes -1 is returned.

- `MLint32 MLTypeGetComponentProperties(char code, MLint32* isSigned, MLint32* isIntegerType, MLint32* isFloatingPointType, MLint32* isLongType);`

  Returns `1 (=true)` in *`*isSigned`*, *`*isIntegerType`*, *`*isFloatingPointType`* and *`*isLongType`* if the component type represented by *`code`* includes this features, otherwise set that flag to `0 (=false)`.

Invalid *code* values return `0 (=false)` in all parameters. It is explicitly permitted to pass NULL as *isSigned*, *isIntegerType*, *isFloatingPointType* or *isLongType* to ignore these. `1 (=true)` is returned if *comp* was a valid component, otherwise the return value is `0 (=false)`.

- `size_t MLTypeComponentSize(char comp);`

  Returns the size of a MLTypeComponent denoted by a character code. On invalid character codes, 0 is returned. Valid codes are:

  - 'b' = bool
  - 'c' = unsigned char
  - 'C' = char
  - 's' = unsigned short
  - 'S' = short
  - 'i' = unsigned int
  - 'I' = int
  - 'l' = unsigned long
  - 'L' = long
  - '6' = MLint64
  - 'f' = float
  - 'd' = double
  - 'D' = long double

- `void MLTypeSetDoubleComponent(char comp, MLdouble val, MLTypeData *dstPtr);`

  Interprets the data referenced by *`*dstPtr`* as data of the type *comp* and sets its value from the passed `MLdouble` value by casting the *`val`* to it. Invalid character codes are ignored.

- `void MLTypeSetIntComponent (char comp, MLCTInt val, MLTypeData *dstPtr);`

  Same as `MLTypeSetDoubleComponent`, but components are set to integer values.

- `void MLTypeSetAllDoubleComponents(const MLTypeInfos *infos, MLdouble val, MLTypeData *dstPtr);`

  All components of the data referenced by *`*dstPtr`* are set to a value cast from the `MLdouble` value *`val`*. Casting is performed by the `MLTypeSetComponent` function.

- `void MLTypeSetAllIntComponent (const MLTypeInfos *infos, MLCTInt val, MLTypeData *dstPtr);`

  Same as `MLTypeSetAllDoubleComponents`, but components are set to integer values.

- `MLdouble MLTypeGetDoubleComponent(char comp, const MLTypeData *dstPtr);`

  Interprets the data referenced by *`*dstPtr`* as data of the type *comp* and returns it as an `MLdouble` value. Invalid character codes are ignored and 0 is returned.

- `MLCTInt MLTypeGetIntComponent (char comp, const MLTypeData *dstPtr);`

  Same as `MLTypeGetComponent`, but components are returned as integer values.

- `void MLTypeShiftLeftComponent(char comp, const MLTypeData *srcPtr, MLCTInt shiftLs, MLTypeData *dstPtr);`

  Interprets the data referenced by *`*dstPtr`* as data of the type *`comp`* and shifts data *`shiftLs`* times to the left, if it is an integer component. Floating point components are multiplied with $2^{shiftLs}$. Negative values for *`shiftLs`* are interpreted as shift right operations or divisions by $2^{shiftLs}$, respectively. Boolean components become `false` on all `shiftLs != 0`. Zero *`shiftLs`* does not change values. Invalid character codes are ignored, i.e., pointers and values are not changed.

7.5.2.2. Convenience Functions to Operate on Registered Voxel Data

Functions to operate on data of registered voxels:

- `MLTypeData *MLAllocateVoxelBuffer(MLDataType dataType, size_t numVoxels, const MLTypeData *voxDefault);`

  Returns a buffer of *`numVoxels`* voxels of data type *`dataType`*. On failure, NULL is returned. If *`voxDefault`* is NULL, the allocated memory is left undefined, otherwise all voxels are filled with the default value pointed to by *`voxDefault`*. The allocated buffer must be removed with `MLFree()`.

- `char *MLGetVoxelValueAsString(const MLTypeData *data, MLDataType dataType, MLErrorCode *errCode);`

  Interprets the data given by *`data`* as a value of type *`dataType`* and returns its value as a string. If anything fails, "" is returned. *`errCode`* may be passed as NULL. If *`errCode`* is not NULL, `*errCode` is set to the error code on failures; otherwise it is set to ML_RESULT_OK. Floating point values are normally printed with maximum precision. The returned pointer must be freed with `MLFree()`.

- `char *MLGetVoxelValueAsStringLimited(const MLTypeData *data, MLDataType dataType, MLErrorCode *errCode, int maxPrec);`

  Interprets the data given by *`data`* as a value of type *`dataType`* and returns its value as a string. If anything fails, "" is returned. *`errCode`* may be passed as NULL. If *`errCode`* is not NULL, `*errCode` is set to the error code on failures; otherwise it is set to ML_RESULT_OK. If *`maxPrec`* is passed with a negative value, the maximum precision of floating point numbers is printed. If passed >= 0, the number of digits is limited to *`maxPrec`*. It will be not larger than the maximum default precision, even when it is accordingly specified. The returned pointer must be freed with `MLFree()`.

- `char *MLTypeComponentsToString(const MLTypeInfos *infos, const MLTypeData *p);`

  Converts a data type instance to a string. *`infos`* point to the type information and *`p`* points to the data of the type instance. The return value is a string containing the type components which are converted to string values that are separated by spaces. It must be freed with `MLFree()`. Floating point values are normally printed with maximum precision. On failures (e.g. *`infos`*==NULL, *`p`*==NULL), an empty string is returned which also must be freed.

- `char *MLTypeComponentsToStringLimited(const MLTypeInfos *infos, const MLTypeData *p, int maxPrec);`

  Converts a data type instance to a string. *`infos`* point to the type information and *`p`* points to the data of the type instance. The return value is a string containing the type components which are converted to string values that are separated by spaces. It must be freed with `MLFree()`. If *`maxPrec`* is passed with a negative value, the maximum precision of floating point numbers is printed. If passed >= 0, the number of digits is limited to *`maxPrec`*. It will not be larger than the maximum default precision even if specified so. On failures (e.g., *`infos`*==NULL, *`p`*==NULL), an empty string is returned which also must be freed.

- `MLint32 MLTypeComponentsFromString(const MLTypeInfos *infos, const char *str, const MLTypeData *defaultVal, MLTypeData *p);`

  Converts a string of a data type instance to instance data, i.e., like an sscanf. *`infos`* point to the type information and *`p`* points to the data of the type instance to be filled with data scanned from the string. The return value is 1 if the string could be scanned successfully. On scan failures or invalid parameters, 0 is returned. If a default value is passed in *`defaultVal`*, components which could not be scanned correctly are copied from their corresponding positions in *`defaultVal`*. If *`defaultVal`* is passed as NULL, those components are left unchanged.

- `MLint32 MLTypeComponentsFromStream(void *iStr, void *iStrStream, void *stdiStr, void *stdiStrStream, const MLTypeInfos *infos, MLTypeData *data);`

  Reads data type components into different stream versions (`istream` and `istrstream` within and outside the standard namespace). Since we have a C interface here, we need to pass the pointers to the streams as `void*` addresses. Hence be careful to which of the first parameters the stream is passed. All other can be set to NULL. On any error, *`*data`* is correctly set as far as possible, and all unreadable values are set to the default value. On bad parameters, failures or not completely readable values, 0 is returned, otherwise 1.

- `MLdouble MLGetVoxelValueAsDouble(const void *data, MLDataType dataType, MLErrorCode *errCode);`

  Interprets the data given by *`data`* as a value of type *`dataType`* and return its value cast to double. If anything fails then 0 is returned. *`errCode`* may be passed as NULL. If *`errCode`* is not NULL then *`*errCode`* is set to the error code on failures; otherwise it is set to `ML_RESULT_OK`.

- `MLCTBool MLTypeCastToBool (const MLTypeInfos *infos, const MLTypeData *p);`

  If *p* is identical to default element, `false (= 0)` is returned, otherwise `true (= 1)`.

- `MLCTInt MLTypeCastToInt (const MLTypeInfos *infos, const MLTypeData *p);`

  The first component of the data type *p* is converted to integer and returned.

- `MLdouble MLTypeCastToDouble (const MLTypeInfos *infos, const MLTypeData *p);`

  The first component of the data type *p* is converted to double and returned.

- `void MLTypeCastFromBool (const MLTypeInfos *infos, MLCTBool p, MLTypeData *q);`

  If `p == 0` then *q* is set to the type default value given by *`infos`*. If `p != 0` then all components of the type are cast to their values cast from 1.

- `void MLTypeCastFromInt (const MLTypeInfos *infos, MLCTInt p, MLTypeData *q);`

  The integer value of *p* is cast to the types of the components and then written to them.

- `void MLTypeCastFromDouble (const MLTypeInfos *infos, MLdouble p, MLTypeData *q);`

  The value *p* is cast to the types of the components and then written to them.

- `void MLTypeBinaryAndInt (const MLTypeInfos *infos, const MLTypeData *p, MLCTInt q, MLTypeData *r);`

  Takes all components from *p* as integer values, applies a bitwise 'and' operation with *q* and writes them as (cast from) integer values back to the corresponding components of *r*.

- `void MLTypeBinaryOrInt (const MLTypeInfos *infos, const MLTypeData *p, MLCTInt q, MLTypeData *r);`

  Takes all components from *p* as integer values, applies a bitwise 'or' operation with *q* and writes them as (cast from) integer values back to the corresponding components of *r*.

- `void MLTypeBinaryXorInt (const MLTypeInfos *infos, const MLTypeData *p, MLCTInt q, MLTypeData *r);`

  Takes all components from *p* as integer values, applies a bitwise 'xor' operation with *q* and writes them as (cast from) integer values back to the corresponding components of *r*.

- `void MLTypeBinaryAnd (const MLTypeInfos *infos, const MLTypeData *p, const MLTypeData *q, MLTypeData *r);`

Takes all components from *p* as integer values, applies a bitwise 'and' operation with corresponding components from *q* (also as integers) and writes them as (cast from) integer values back to the corresponding components of *r*.

- `void MLTypeBinaryOr (const MLTypeInfos *infos, const MLTypeData *p, const MLTypeData *q, MLTypeData *r);`

  Takes all components from *p* as integer values, applies a bitwise 'or' operation with corresponding components from *q* (also as integers) and writes them as (cast from) integer values back to the corresponding components of *r*.

- `void MLTypeBinaryXor (const MLTypeInfos *infos, const MLTypeData *p, const MLTypeData *q, MLTypeData *r);`

  Takes all components from *p* as integer values, applies a bitwise 'xor' operation with corresponding components from *q* (also as integers) and writes them as (cast from) integer values back to the corresponding components of *r*.

- `void MLTypeShiftComponentsLeft(const MLTypeInfos *infos, const MLTypeData *p, MLCTInt q, MLTypeData *r);`

  Takes one data type component after another and shifts each component left `shiftLs` times if it is an integer component. Floating point components are multiplied with $2^{\texttt{shiftLs}}$. Negative values for `shiftLs` are interpreted as shift right operations or divisions by $2^{\texttt{shiftLs}}$, respectively. Boolean components become `false` on all `shiftLs != 0`. Zero `shiftLs` does not change any component.

- `void MLTypeCastToOtherType(const MLTypeInfos *otherInfos, const MLTypeData *otherData, const MLTypeInfos *myInfos, MLTypeData *myData);`

  Converts a data instance referenced by *otherData* of a type specified by *otherInfos* to another data instance referenced by **myData* of a type specified by *myInfos*. As long as components of any data type in the source exist, the *myData* components are set to the same values. Components which do not have a counterpart in the *otherData* are filled with the counterparts from its default value given by the *myInfos*. E.g.: If an (int, char, double) data type (represented by "ICd") is cast to a four component float vector (represented by "ffff"), then the first three components are set from an int cast to double, from an char cast to double and from an double cast to double. The fourth component is copied from the fourth component of the type default value given in the *dstInfos* of type `MLTypeInfo`.

- `void MLTypeCastFromOtherType(const MLTypeInfos *otherInfos, const MLTypeData *otherData, const MLTypeInfos *myInfos, MLTypeData *myData);`

  Casts another data element *otherData* with attributes given by *otherInfos* to *myData* of a type given by *myInfos*. See `MLTypeCastToOtherType` for more infos.

- `MLint32 MLTypeIsEqualToOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. If that buffer is equal to *myData* then `1 (=true)` is returned, otherwise `0 (=false)`. For the comparison `myInfos->isEqualToType` is used.

- `MLint32 MLTypeIsSmallerThanOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. If that buffer is smaller than *myData* then `1 (=true)` is returned, otherwise `0 (=false)`. For the comparison `myInfos->isSmallerThanType` is used.

- `MLint32 MLTypeIsGreaterThanOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData);`

Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. If that buffer is greater to *myData* then `1 (=true)` is returned, otherwise `0 (=false)`. For the comparison `myInfos->isGreaterThanType` is used.

- `void MLTypeMultWithOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData, MLTypeData *r);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. That buffer is multiplied with *myData* and written into *r*. For the multiplication `myInfos->multWithType` is used.

- `void MLTypeDivByOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData, MLTypeData *r);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. Then *myData* is divided by the buffer and written into *r*. For the division `myInfos->divByType` is used.

- `void MLTypeAddOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData, MLTypeData *r);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. That buffer is added with *myData* and written into *r*. For the addition `myInfos->addToType` is used.

- `void MLTypeSubtractOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData, MLTypeData *r);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. That buffer is subtracted from *myData* and written into *r*. For the subtraction `myInfos->subtractFromType` is used.

- `void MLTypePowerOfOtherType(const MLTypeInfos *myInfos, const MLTypeData *myData, const MLTypeInfos *otherInfos, const MLTypeData *otherData, MLTypeData *r);`

  Casts another data element *otherData* with attributes given by *otherInfos* to a local buffer of a type given by *myInfos*. The power of *myData* with the buffer is calculated and written into *r*. For the power calculation `myInfos->powerOfType` is used.

7.5.3. Reducing Generated Code and Compile Times

Sometimes a module programmer knows that a module only makes sense for images with certain voxel types. In this case, the number of potential voxel types can be reduced so that the code is smaller and the compilation times are shortened.

Typical application areas are binary operations on voxels which only work fine on integer voxels; or operations on normalized values which are always between 0 and 1 and consequently require floating point type voxels to avoid information loss. Also, some operations such as gradient calculations or tensor imaging might require operations which only make sense on registered vector or matrix voxels.

A typical ML module uses a `ML_CALCULATE_OUTPUTSUBIMAGE` macro to compile the template `calculateOutputSubImage` function for all scalar types:

```
// Implements the call  to the typed calculateOutputSubImage method for all potential data types.
ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_SCALAR_TYPES_CPP(NormalModule);

//! Fill output page with calculated data in a module with one input.
template <typename DATATYPE>
void NormalModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                   int outIndex, TSubImage<DATATYPE> *inSubImg)
{
  // Calculate contents of outSubImg here.
}
```

The following example shows how to compile the template function for all available integer types only. It uses a special `ML_CALCULATE_OUTPUTSUBIMAGE` macro which accepts an additional parameter to determine the set of data type cases to be compiled:

```
// Implements the call  to the typed calculateOutputSubImage method for all integer types.
ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_WITH_CUSTOM_SWITCH_CPP(CalcTest, ML_IMPLEMENT_INT_CASES);

//! Fill output page with calculated data in a module with one input.
template <typename DATATYPE>
void NormalModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                   int outIndex, TSubImage<DATATYPE> *inSubImg)
{
  // Calculate contents of outSubImg here.
}
```

The following predefined type configurations of data type cases can be used:

- `ML_IMPLEMENT_INT_CASES` - Implements all integer types.
- `ML_IMPLEMENT_FLOAT_CASES` - Implements all floating point types.
- `ML_IMPLEMENT_INT_FLOAT_CASES` - Implements all integer and floating point types.
- `ML_IMPLEMENT_INT_FLOAT_CASES_WO_INT64` - Implements all integer and floating point types without the 64 bit integer types.
- `ML_IMPLEMENT_COMPLEX_CASES` - Implements the complex types.
- `ML_IMPLEMENT_VECTOR_CASES` - Implements the default vector types: `Vector2`, `Vector3` and `Vector6` (both with float and double component types).
- `ML_IMPLEMENT_MATRIX_CASES` - Implements the default matrix types: `Matrix2` and `Matrix3` (both with float and double component types).
- `ML_IMPLEMENT_SCALAR_CASES` - Identical to `ML_IMPLEMENT_INT_FLOAT_CASES` which implements the scalar voxel types used in most ML modules.
- `ML_IMPLEMENT_DEFAULT_CASES` - This combines `ML_IMPLEMENT_SCALAR_CASES`, `ML_IMPLEMENT_COMPLEX_CASES`, `ML_IMPLEMENT_VECTOR_CASES` and `ML_IMPLEMENT_MATRIX_CASES`.

You can also configure your own combinations using the following constants:

- `ML_IMPLEMENT_INT8_CASE` and `ML_IMPLEMENT_UINT8_CASE` - Signed and unsigned 8 bit integers.
- `ML_IMPLEMENT_INT16_CASE` and `ML_IMPLEMENT_UINT16_CASE` - Signed and unsigned 16 bit integers.
- `ML_IMPLEMENT_INT32_CASE` and `ML_IMPLEMENT_UINT32_CASE` - Signed and unsigned 32 bit integers.
- `ML_IMPLEMENT_INT64_CASE` - Signed 64 bit integer.
- `ML_IMPLEMENT_FLOAT_CASE`, `ML_IMPLEMENT_DOUBLE_CASE` - Floating point types.

In the following example only the `MLint64`, the `MLdouble` and all complex types are compiled:

```
#define ML_IMPLEMENT_LARGE_AND_COMPLEX_CASES(CLASS_NAME, SWITCH_CODE, DUMMY1, DUMMY2, DUMMY3)  \
    ML_IMPLEMENT_INT64_CASE(     CLASS_NAME, SWITCH_CODE, DUMMY1, DUMMY2, DUMMY3) \
    ML_IMPLEMENT_DOUBLE_CASE(    CLASS_NAME, SWITCH_CODE, DUMMY1, DUMMY2, DUMMY3) \
    ML_IMPLEMENT_COMPLEX_CASES(  CLASS_NAME, SWITCH_CODE, DUMMY1, DUMMY2, DUMMY3)

  ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_WITH_CUSTOM_SWITCH_CPP
    (CalcTest, ML_IMPLEMENT_LARGE_AND_COMPLEX_CASES);

  template <typename DATATYPE>
    void NormalModule::calculateOutputSubImage(TSubImage<DATATYPE> *outSubImg,
                                       int outIndex,
                                       TSubImage<DATATYPE> *inSubImg){ ... }
```

7.5.4. Configuration of Supported Voxel Types

There are some advanced options you can use when you activate the support of your ML module for registered data types.

- `setVoxelDataTypeSupport(ONLY_DEFAULT_TYPE)` lets the module work with the default type set supported by macros like `ML_CALCULATEOUTPUTSUBIMAGE_NUM_INPUTS_1_DEFAULT_TYPES_CPP`. So if you use a module macro with `DEFAULT` in its name, use this value.
- `setVoxelDataTypeSupport(ONLY_SCALAR_TYPES)` is the default setting which causes the ML to deactivate all modules outputs if any of the voxel data is of non-scalar type. This stops a module from operating on non-scalar data types, so that a module programmer does not have to care about them at all.
- `setVoxelDataTypeSupport(ALL_REGISTERED_TYPES)` is the mode to have the module work with all registered voxel types which do not provide all operations of a standard type.

Note

A module can always restrict the types it supports in the `calculateOutputImageProperties` method by setting an output to invalid if a type combination is not supported.

This mode is useful for modules which handle or just pass voxel data, but do not calculate explicit output values, like for example a `SubImage` module. It is also useful for modules which support types other than the scalar types when they are not using the module macro for the default type set, e.g. when a module is implemented through typed output handlers.

7.5.5. Implementing a New Voxel Data Type by Deriving from MLTypeInfos

You can register your own voxel data type. A structure that describes the data type, its properties and a function table with its operations can be registered in the ML to activate a new type. Modules that perform generic operations that use the registered type structures (directly or indirectly) will automatically work with this new type.

There are some steps to take and many functions to implement, but generally it is not really difficult and does not involve as much work as one might think. The easiest way is to use the `MLTypeAddExampleInfos` example as a template for integrating a new type; this way you will not forget any important steps. See MLTypeAddExample for detailed information.

1. Take an `MLTypeInfos` structure from `mlTypeDefs.h` .
2. Set all function pointers in that structure to functions that implement your data type operations.
3. Initialize your `MLTypeInfos` (Section 7.5.5.1, “Describing a New Voxel Type with MLTypeInfos”) (see `mlTypeDefs.h`) by using `MLTypeInfosInit()` (see `mlDataTypes.h`).
4. Create one instance of your initialized `MLTypeInfos` and register it on library initialization by using `MLRegisterTypeInfos()` from `mlDataTypes.h`.

The result of your implementation should be an initialized `MLTypeInfos` structure that describes all data type properties, features and operations (see Section 7.5.5.1, “Describing a New Voxel Type with MLTypeInfos”). Take a closer look at this structure now.

7.5.5.1. Describing a New Voxel Type with MLTypeInfos

The `MLTypeInfos` structure describes all features, properties and operations of a data type. It contains all data type features and pointers to all functions needed to implement generic operations on the data type. It is a wrapper for any information and code needed for any standard or user-defined data type the ML uses. The descriptions of the scalar ML data types are implemented in .

The descriptive components of the `MLTypeInfos` structure are permanent and non-changing values requested by many operations that need data type information. Most of these values can be initialized with the function `MLTypeInfosInit()` which also performs checks for valid data type initialization and calculates the more difficult components of the `MLTypeInfos` structure. All other stuff (i.e., the function pointers) should be initialized by using the macro `ML_TYPE_ASSIGN_FUNCTION_POINTERS()` as explained in the example in Section 7.5.5.2, “The MLTypeAddExample”.

1. *`size_t numComps`*

   Number of components of this data type. Equals number of characters in *`structInfoString`*. A scalar value has 1 component, complex numbers have 2 components, and ML vectors have 6 components (see Section 2.4.1, “`ImageVector`, ImageVector” for details) Each component must be a scalar object as described in number 13 [148].

2. *`size_t typeSize`*

   The `sizeof` of the registered data type, i.e., its size in bytes.

3. *`const char *name`*

   The pointer to a null-terminated character string that gives the data type name. It should contain alphanumeric characters only.

4. *`MLDataType rangeAndPrecisionEquivalent`*

   Returns a standard data type which has a comparable range and precision behavior.

5. `double dblMin`

   Double minimum of data type.

6. `double dblMax`

   Double maximum of data type.

7. *`const MLTypeData *typeMinPtr`*

   Minimum value of the data type.

8. *`const MLTypeData *typeMaxPtr`*

   Maximum value of the data type.

9. *`const MLTypeData *typeDefaultPtr`*

   The default value of the data type, comparable to zero.

10. *`size_t numGoodCastTos`*

    Number of data types to which this type can be cast without information or functionality loss.

11. *`const char **goodCastTos`*

    Pointer to a table of a null-terminated string of data type names to which this type can be cast without information or functionality loss.

12. *`unsigned int compOffsets[ML_MAX_COMPONENTS_EXTENDED_TYPE]`*

    Table of byte offsets from the first component to other components to directly address any component with a character pointer. e.g., if a data type consists of a float, a char and another float component where `sizeof(float)` is 4 and `sizeof(char)` is 1, the first table entry must be 0, the second entry must be 4, and the third entry must be 5.

13. *`const char *structInfoString`*

    Pointer to a null-terminated string that describes the type configuration as explained for the *`typeStructInfo`* parameter of the function `MLTypeInfoInit` .

14. *`int dataTypeId`*

    The `MLDataType` id of the registered type. This should be a constant value. If you want to define your own types you should contact the MeVisLab team to get your own id range assigned.

The operative components of the `MLTypeInfos` structure are function pointers which are called when operations on a registered data type are needed. We will forgo the opportunity to list all functions here, simply refer to the definition of `MLTypeInfos` in `mlTypeDefs.h` for the required functions.

Many operations can simply be implemented by using convenience functions which are already implemented in the ML, e.g. to cast one extended data type to another.

The parameters of these functions are often pointers of type `MLTypeData` to instances of the data type; the parameters need to be cast to be able to work on the correct data type.

The `MLTypeInfosInit()` function checks for valid data type initialization, and calculates the more difficult components of the `MLTypeInfos` structure. It returns `1(=true)` on success, `0(=false)` on failure.

Please refer to the `MLTypeAddExample` example on how exactly to register your own type.

7.5.5.2. The MLTypeAddExample

The following example shows how to implement a new voxel type. The example does not implement all functions in order to keep the example short. However, the implementation of most functions should not be a problem when you look at similar functions for reference.

Note

- Many functions are implemented by using ML functions; they typically implement the desired operation for each components of the new data type. Thus, especially vector operations can often be implemented easily. See the header file documentation of those functions for detailed descriptions.
- Most functions get pointers to the data instances by `const MLTypeData` or `MLTypeData` pointers. This is necessary because the functions are defined generically and don't know the real type. Thus, many casts of those pointers are needed before the actual type operations can be applied.
- Do not change the function names because it is exactly these names that are used in the `ML_TYPE_ASSIGN_FUNCTION_POINTERS()` macros to set the members in the `MLTypeInfos` structure. Thus, missing functions will also be detected which makes sure that no function is forgotten.

The initialization of the new voxel data type, typically to be implemented in the library initialization file:

Example 7.9. `MLRegisterTypeInfos`

```
//! Create static instances of all data types to be used in the ML.
//! These instances will directly be registered as new ML data types.
static MLTypeAddExampleInfos _MLNewVType;
int initResult = MLRegisterTypeInfos(&_MLNewVType);
return initResult;
```

The implementation of the `MLTypeAddExampleInfos` class which is used to create the registered instance in the library initialization file:

Example 7.10. How to Add Your Own Voxel Data Type

This example is outdated. Please refer to the example code provided with the MeVisLab SDK for the current version.

```
#ifndef __mlMLGuideTypeAddExampleInfos_H
#define __mlMLGuideTypeAddExampleInfos_H

// ML-includes
#ifndef __MLTypeAddExampleSystem_H
#include "MLTypeAddExampleSystem.h"
#endif
#ifndef __mlDataTypes_H
#include "mlDataTypes.h"
#endif
#ifndef __mlUtils_H
#include "mlUtils.h"
#endif
```

```
ML_START_NAMESPACE

//! The data type to be implemented as a new voxel data type.
//! For simplification we register a new type which does the
//! same as the normal MLdouble type.
typedef MLdouble NewVType;

//----------------------------------------------------------------------------------
//! Example class to create a new voxel data types to be registered in the ML.
//----------------------------------------------------------------------------------
class ML_NEW_VTYPEEXTENSION_EXPORT MLTypeAddExampleInfos : public MLTypeInfos {

protected:
```

```
  //! Reference to a permanently existing constant instance of NewVType containing the
  //! minimum data type value.
  static const NewVType &_typeMin()    { static NewVType v=-DBL_MAX; return v; }

  //! Reference to a permanently existing constant instance of NewVType containing the
  //! maximum data type value.
  static const NewVType &_typeMax()    { static NewVType v=DBL_MAX;  return v; }

  //! Reference to a permanently existing constant instance of NewVType containing the
  //! default data type value.
  static const NewVType &_typeDefault(){ static NewVType v=0;        return v; }

  //! Permanent instance of a pointer to the typeInfos used by this class. It
  //! will often be used as a kind of this pointer for the static instance of
  //! this data type information.
  static MLTypeInfos *_myInfos;

  //! Number of instances of this class. Only used to avoid that more than one
  //! instance is created.
  static size_t _numInstances;
```

```
public:

  //! Constructor. It initializes
  //! - all data type operations by setting pointers of a
  //!   function table to the data type operations (implemented
  //!   as static functions) by using the macro
  //!   ML_TYPE_ASSIGN_FUNCTION_POINTERS();
  //! - all other data type properties, like min/max/default values (as
  //!   MLdouble and as type values) by using the function MLTypeInfosInit(),
  //! - it checks for at most one instance of this class.
  MLTypeAddExampleInfos()
  {
    // We permit only one instance since most class settings are static constant.
    if (_numInstances > 0){
      mlError("MLTypeAddExampleInfos", ML_PROGRAMMING_ERROR)
        << "Too many instances of MLTypeAddExampleInfos created.";
    }
    _numInstances++;

    // Store pointer to this. We only have one instance. So we simulate a kind
    // "this" pointer for this static instance.
    _myInfos = this;

    // Assign all pointers to the static functions implementing the operations.
    // The function names have predefined names beginning with "MLTYPE_" (see
    // function implementation below).
    ML_TYPE_ASSIGN_FUNCTION_POINTERS();
```

```
    // Specify all type names and their number to which this type can be cast
    // without information loss.
    size_t numGoodCastTos = 1;
    static const char *goodCastTos[] = { "NewVType" };

    // Initialize the new MLTypeInfos struct. For a parameter description see
    // discussion of MLTypeInfos structure or the type documentation in the
    // mlTypeDefs.h file.
    NewVType buf;
    void *addr[1];
    addr[0] = &buf;
    MLTypeInfosInit(this,
                    sizeof(NewVType),
                    "NewVType",
                    -DBL_MAX,
                    DBL_MAX,
                    (MLTypeData*)(&_typeMin()),
                    (MLTypeData*)(&_typeMax()),
                    (MLTypeData*)(&_typeDefault()),
                    "d",
                    false,
                    MLdoubleType,
                    addr,
                    numGoodCastTos,
                    goodCastTos
                    );
  }
```

```
  //! Return value as string to be freed by MLFree().
  //! Use MLTypeComponentsToString() if possible.
  static char *MLTYPE_getStringValue(const MLTypeData *p)
  { return MLTypeComponentsToString(_myInfos, p); }

  //! Convert string s to value and write result into r.
```

```
//! Use MLTypeComponentsFromString() if possible.
static void MLTYPE_setStringValue(const char *s, MLTypeData *r)
{ MLTypeComponentsFromString(_myInfos, s, (MLTypeData*)&(_typeDefault()), r);  }

// IMPLEMENT MINIMUM/MAXIMUM/DEFAULT AND COPY OPERATIONS.
//! Sets p to minimum value. Must be implemented.
static void MLTYPE_setToMinimum(MLTypeData *p)
{ memcpy(p, &_typeMin(), sizeof(NewVType)); }
```

```
//! Sets p to minimum value. Must be implemented.
static void MLTYPE_setToMaximum(MLTypeData *p)
{ memcpy(p, &_typeMax(), sizeof(NewVType)); }

//! Sets p to default value. Must be implemented.
static void MLTYPE_setToDefault(MLTypeData *p)
{ memcpy(p, &_typeDefault(), sizeof(NewVType)); }

//! Copy parameter p to second q.
static void MLTYPE_copy(const MLTypeData *p, MLTypeData *q)
{ memcpy(q, p, sizeof(NewVType)); }
```

```
// IMPLEMENT CAST OPERATIONS FROM THE NEW TYPE TO BOOL/INT/DOUBLE/OTHER TYPE.
//! Return parameter p cast to bool. Typically false when it is identical to
//! the default element, otherwise true.
static MLCTBool MLTYPE_castToBool(const MLTypeData *p)
{ return (*((NewVType*)p)) != _typeDefault(); }

//! Return parameter p cast to integer. Often implemented as
//! the integer cast of the first component.
static MLCTInt MLTYPE_castToInt(const MLTypeData *p)
{ return    (MLCTInt)( *((NewVType*)p) ); }

//! Return parameter p cast to MLdouble. Often implemented as
//! the integer cast of the first component.
static MLdouble MLTYPE_castToDouble(const MLTypeData *p)
{ return (MLdouble)( *((NewVType*)p) ); }

//! Cast myData to otherData who has type infos otherInfos. Usually
//! implemented by default with function casting componentwise.
static void MLTYPE_castToOtherType(const MLTypeData *myData,
                                   const MLTypeInfos *otherInfos,
                                   MLTypeData *otherData)
{ MLTypeCastToOtherType(_myInfos, myData,  otherInfos, otherData); }
```

```
// IMPLEMENT CAST OPERATIONS FROM INT/DOUBLE/OTHER TYPE TO THE NEW TYPE.
//! Cast first parameters to data type and write it into second parameter.
static void MLTYPE_castFromInt(MLCTInt p, MLTypeData *q)
{ *((NewVType*)q) = (NewVType)p;  }

//! Cast first parameters to data type and write it into second parameter.
static void MLTYPE_castFromDouble(MLdouble p, MLTypeData *q)
{ *((NewVType*)q) = (NewVType)p;  }

//! Cast first parameters to data type and write it into second parameter.
static void MLTYPE_castFromOtherType(const MLTypeInfos *otherInfos,
                                     const MLTypeData *otherData,
                                     MLTypeData *myData)
{ MLTypeCastToOtherType(otherInfos,otherData, _myInfos, myData); }
```

```
static MLCTBool MLTYPE_isEqualToType(const MLTypeData *p, const MLTypeData *q)
{ return (*((NewVType*)p)) == (*((NewVType*)q)); }
```

```
// IMPLEMENT SOME SPECIAL FUNCTIONS
//! Negate the value.
static void MLTYPE_negate(const MLTypeData *p, MLTypeData *q)
{ *((NewVType*)q) = -(*((NewVType*)p)); }

//! Normalize type.
static void MLTYPE_normalize (const MLTypeData * /*p*/, MLTypeData *q)
{ *((NewVType*)q) = (NewVType)(1); }
```

```
// IMPLEMENT MULTIPLICATION FUNCTIONS. THE RESULT IS WRITTEN ALWAYS INTO LAST
// FUNCTION PARAMETER R.
//! Implement multiplication with integer. Result written into parameter three.
static void MLTYPE_multWithInt(const MLTypeData *p, MLCTInt q, MLTypeData *r)
{ *((NewVType*)r) = (*((NewVType*)p)) * (NewVType)q; }

//! Implement multiplication with double. Result written into parameter three.
static void MLTYPE_multWithDouble(const MLTypeData *p, MLdouble q, MLTypeData *r)
{ *((NewVType*)r) = (*((NewVType*)p)) * (NewVType)q; }

//! Implement multiplication with its own type. Result written into parameter three.
static void MLTYPE_multWithType(const MLTypeData *p, const MLTypeData *q, MLTypeData *r)
```

```
 { *((NewVType*)r) = (*((NewVType*)p)) * (*((NewVType*)q)); }

//! Implement multiplication with another type. Result written into parameter three.
static void MLTYPE_multWithOtherType(const MLTypeInfos *otherInfos,
                                     const MLTypeData  *otherData,
                                     const MLTypeData  *myData,
                                     MLTypeData        *r)
{ MLTypeMultWithOtherType(_myInfos, myData, otherInfos, otherData, r); }
```

```
//! IMPLEMENT ADDITIONS. SEE MULTIPLICATION FUNCTIONS FOR SIMILAR CODE.
static void MLTYPE_plusInt   (const MLTypeData *p, MLCTInt q, MLTypeData *r)  { /*...*/ }
static void MLTYPE_plusDouble(const MLTypeData *p, MLdouble q, MLTypeData *r) { /*...*/ }
static void MLTYPE_plusType  (const MLTypeData *p, const MLTypeData *q, MLTypeData *r) { /*...*/ }
```

Chapter 8. Base Objects

Chapter Objectives

This chapter contains all the information you need to implement persistence to non-module classes in the ML. Many modules and classes provide special functionality to handle objects derived from the class `Base` or `TreeNode` so that they can often handle objects they do not even know explicitly (see Section 8.1, "`Base` Objects" and Section 8.3, "Creating Trees from `Base` Objects Using TreeNodes").

This chapter explains how

- to derive your own objects from `Base`,
- these objects can be stored and retrieved in/from trees (e.g. as XML or `RawNode` trees) with `TreeNode` (see Section 8.3, "Creating Trees from `Base` Objects Using TreeNodes").
- they can be written to or read from an `AbstractPersistenceStream` (see Section 8.4, "Writing/Reading `Base` Objects to/from AbstractPersistenceStream"). This mechanism is intended to replace the `TreeNode` persistence completely in the future.
- they can be (de)composed with other `Base` objects to larger structures and
- they can be stored and retrieved in files by using already existing ML modules dedicated to that.

See Section 2.1.2.3, "`Base` Field" for information on how to derive your own class from `Base` and how to transfer `Base` objects between modules.

8.1. Base Objects

When you want to include a new class to the ML that is not an ML module (e.g., to pass additional image or segmentation information from one module to another), the `Base` persistence mechanism of the ML should be used. It permits saving and storing objects, passing objects from one module to another `BaseField` or simply getting and setting their state via strings. This class represents general ML objects that support import/export via strings (`setPersistentState()`/`persistentState()`) or arbitrary tree structures (using `addStateToTree()` and `readStateFromTree()`) or through a specialized input/output stream (using `writeTo()` and `readFrom()`). It has to be the base class for all objects passed from one `BaseField` to another (see Section 2.1.2, “ Field ”).

Note

This class is the base class for the class `Module` and all derived modules.

- It can be represented by using the field concept (class `BaseField`)
- It provides an interface to allow for the import/export of a persistent representation of an object's internal state.

8.2. Composing, Storing and Retrieving Base Objects

`Base` objects can be composed and decomposed to lists of type `BaseList`. This functionality is provided by the modules `ComposeBaseList` and `DecomposeBaseList`. See the documentation of these modules for details.

`Base` objects that support the `TreeNode` persistence mechanism can be stored and restored using the `SaveBase` and `LoadBase` modules.

8.3. Creating Trees from Base Objects Using TreeNodes

To create a class of Base objects that supports persistence and that can be stored and restored using the `SaveBase` and `LoadBase` modules (see Section 8.2, “Composing, Storing and Retrieving `Base` Objects”), the following steps need to be taken:

- Derive your custom class from `Base` or another class derived from `Base` (Section 2.1.2.3, “`Base` Field”).
- Include `mlTreeNode.h` in your header file.
- Overwrite the virtual methods `addStateToTree()` and `readStateFromTree()`.
- Assign a version number to your class by using the macro `ML_SET_ADDSTATE_VERSION(VersionNumber)` in your public class header.
- Add the `ML_CLASS_HEADER(ClassName)` macro in the header.
- Add the `ML_CLASS_SOURCE(ClassName, SuperClassName)` macro in the cpp.
- Call `YourClass::initClass()` in the project's init.cpp file.

The following example shows how to implement persistence to a simple class `SegmentedObject`. The class `SegmentedObject` is derived from `BaseItem` which is derived from `Base`:

```
Base (abstract class, no members)
     |
BaseItem (name, id)
     |
SegmentedObject (objectGrayValue, voxelCount, boundingBox)
```

Example 8.1. How to Implement Persistence for `Base` Objects

Header file of a class `SegmentedObject`:

```
class SegmentedObject : public BaseItem {

...

public:
      //! Implement export functionality (as used by the SaveBase module):
      virtual void addStateToTree(TreeNode* parent) const;

      // Set current version number
      ML_SET_ADDSTATE_VERSION(1);

      //! Implement import functionality (as used by the LoadBase module):
      virtual void readStateFromTree(TreeNode* parent);

private:
      //! ML runtime system related stuff
      ML_CLASS_HEADER(SegmentedObject);

      // Members to be (re-)stored:
      //! The identifying gray value of this object
      long _objectGrayValue;

      //! Number of voxels
      long _voxelCount;

      //! Bounding box respective to original image
      SubImageBox* _boundingBox;

...

}
```

Source file of class `SegmentedObject`:

Adding the state to the tree:

```
//! Implement export functionality:
void SegmentedObject::addStateToTree(TreeNode* parent) const
{
  // Write version number (as set in the header)
  ML_ADDSTATE_VERSION(SegmentedObject);

  // Add superclass members:
  ML_ADDSTATE_SUPER(BaseItem);

  // Add this class' members:
  parent->addChild(_objectGrayValue, "ObjectGrayValue");
  parent->addChild(_voxelCount, "VoxelCount");

  // The bounding box is optional, do not write if the pointer is NULL:
  if (_boundingBox) { parent->addChild(*_boundingBox, "BoundingBox"); }
}
```

Reading the state from the tree:

```
//! Implement import functionality:
void SegmentedObject::readStateFromTree(TreeNode* parent)
{
  // Read version number
  int version = parent->getVersion("SegmentedObject");

  // Read super class members:
  ML_READSTATE_SUPER(BaseItem);

  // Handle version differences:
  // In this example, version 0 used a different tag for _objectGrayValue
  // and did not write the VoxelCount value.
  switch (version) {
    case 0 :
      // Read object gray value from old tag name
```

```
      parent->readChild(_objectGrayValue, "GrayValue");
      break;
    case 1 :
      parent->readChild(_objectGrayValue, "ObjectGrayValue");
      break;
    default:
      // Throw exception: A version upgrade was performed without adapting the version handling
      throw TreeNodeException(TNE_UnsupportedClassVersion);
  }

  // Handle this version difference (voxelCount available or not)
  // by calling the macro ML_READCHILD_OPTIONAL which sets
  // the given variable to a default value (third parameter)
  // in case the tag "VoxelCount" was not found.
  ML_READCHILD_OPTIONAL(_voxelCount, "VoxelCount", 0);

  // Bounding box is optional:
  // However, ML_READCHILD_OPTIONAL is not designed for objects references,
  // hence we have to handle the case manually:
  if (!_boundingBox) { ML_CHECK_NEW(_boundingBox, new SubImageBox()); }
  try {
    parent->readChild(*_boundingBox, "BB");
  }
  catch (const TreeNodeException& e) {
    // Some other exception? Pass problem to caller.
    if (e.getCode() != TNE_ChildNotFound) { throw; }

    // No, a Child Not Found exception occurred, we handle it manually:
    ML_DELETE(_boundingBox);
  }
}
```

Registering the class in the runtime type system:

```
  ML_CLASS_SOURCE(SegmentedObject, BaseItem)
```

8.4. Writing/Reading `Base` Objects to/from AbstractPersistenceStream

To create a class of Base objects that supports persistence and that can be stored and restored using the `SaveBase` and `LoadBase` modules (see Section 8.2, “Composing, Storing and Retrieving `Base` Objects”), the following steps can be taken (this is an alternative to the `TreeNode` persistence mechanism):

- Derive your custom class from `Base` or another class derived from `Base` (Section 2.1.2.3, “`Base` Field”).
- Include `mlAbstractPersistenceStream.h` in your header file.
- Overwrite the virtual methods `writeTo()` and `readFrom()`.
- Overwrite the virtual method `implementsPersistence()` to return true for the persistence interface(s) that you implement. This is a new requirement so that other instances can decide which persistence interface to use.
- Assign a version number to your class by using the macro `ML_SET_ADDSTATE_VERSION(VersionNumber)` in your public class header.

  This is the same as in the `TreeNode` interface
- Add the `ML_CLASS_HEADER(ClassName)` macro in the header.
- Add the `ML_CLASS_SOURCE(ClassName, SuperClassName)` macro in the cpp.
- Call `YourClass::initClass()` in the project's init.cpp file.

The following example shows how to implement persistence to a simple class `SegmentedObject`. The class `SegmentedObject` is derived from `BaseItem` which is derived from `Base`:

```
Base (abstract class, no members)
```

```
     |
BaseItem (name, id)
     |
SegmentedObject (objectGrayValue, voxelCount, boundingBox)
```

Example 8.2. How to Implement Persistence for `Base` Objects

Header file of a class `SegmentedObject`:

```
class SegmentedObject : public BaseItem {

...

public:
      //! announce supported persistence interfaces
      virtual bool implementsPersistence(PersistenceInterface iface) const
      {
          return (iface == PersistenceByStream);
      }

      //! Implement export functionality (as used by the SaveBase module):
      virtual void writeTo(AbstractPersistenceOutputStream* stream) const;

      // Set current version number
      ML_SET_ADDSTATE_VERSION(1);

      //! Implement import functionality (as used by the LoadBase module):
      virtual void readFrom(AbstractPersistenceInputStream* stream, int version);

private:
      //! ML runtime system related stuff
      ML_CLASS_HEADER(SegmentedObject);

      // Members to be (re-)stored:
      //! The identifying gray value of this object
      long _objectGrayValue;

      //! Number of voxels
      long _voxelCount;

      //! Bounding box respective to original image
      SubImageBox* _boundingBox;

...

}
```

Source file of class `SegmentedObject`:

Writing the object state to the stream:

```
//! Implement export functionality:
void SegmentedObject::writeTo(AbstractPersistenceOutputStream* stream) const
{
  // Add superclass members:
  ML_WRITETO_SUPER(BaseItem, stream);

  // Add this class' members:
  stream->write(_objectGrayValue, "ObjectGrayValue");
  stream->write(_voxelCount, "VoxelCount");

  // The bounding box is optional, do not write if the pointer is NULL:
  if (_boundingBox) {
    // start a new sub-structure
    stream->startStruct("BoundingBox");
    stream->write(_boundingBox->v1, "v1");
    stream->write(_boundingBox->v2, "v2");
    stream->endStruct();
  }
}
```

Reading the object state from the stream:

```
//! Implement import functionality:
void SegmentedObject::readFrom(AbstractPersistenceInputStream* stream, int version)
{
  // Read super class members:
  ML_READFROM_SUPER(BaseItem, stream);

  // Handle version differences:
```

```
  // In this example, version 0 used a different tag for _objectGrayValue
  // and did not write the VoxelCount value.
  switch (version) {
    case 0 :
      // Read object gray value from old tag name
      parent->read(_objectGrayValue, "GrayValue");
      break;
    case 1 :
      parent->read(_objectGrayValue, "ObjectGrayValue");
      break;
    default:
      // Throw exception: A version upgrade was performed without adapting the version handling
      // Note that this exception only needs to be thrown if you want to be on the safe side.
      // The persistence framework outputs a warning on its own if a newer version than that
      // from ML_SET_ADDSTATE_VERSION is encountered.
      throw PersistenceStreamFormatException("Unsupported version");
  }

  // Handle version difference (voxelCount available or not)
  // by calling the macro readOptional which sets
  // the given variable to a default value (second parameter)
  // in case the tag "VoxelCount" was not found.
  stream->readOptional(_voxelCount, 0, "VoxelCount");

  // Bounding box is optional:
  // However, startStruct can not be called optionally,
  // hence we have to check beforehand if there is an element with the correct name:
  if (stream->isNextInStruct("BoundingBox")) {
    try {
      ML_CHECK_NEW(_boundingBox, new SubImageBox());
      stream->startStruct("BoundingBox");
      stream->read(_boundingBox->v1, "v1");
      stream->read(_boundingBox->v2, "v2");
      stream->endStruct()
    }
    catch (const PersistenceStreamException& e) {
      // make sure to delete bounding box again:
      ML_DELETE(_boundingBox);
      // re-throw exception
      throw;
    }
  }
}
```

Registering the class in the runtime type system:

```
  ML_CLASS_SOURCE(SegmentedObject, BaseItem)
```

Chapter 9. Unicode Support

Chapter Objectives

This chapter describes features and limitations of the ML with regard to international character handling.

9.1. Unicode Support

Unicodes must be supported to handle international characters in character strings such as parameters or file names. The ML implements support of the so-called UTF8 unicodes and provides some functions for managing or recoding these UTF8 unicodes. Thus, all strings (especially field names) the ML handles may contain unicoded characters.

Important

All obtained strings from e.g., module fields or other ML sources may contain unicoded characters.

Strings (e.g., file names) must be handled with I/O functions that are both capable of dealing with unicode and platform-independent.

See Section 2.6.2, “ `MLUtilities` ” and Chapter 10, *File System Support* for more information on helper functions for the platform-independent implementation of unicode-related stuff.

Note

When you receive other unicoded strings (e.g., from user interfaces, other libraries or from string files), these strings might use other uni-codings. See `mlUnicode.h` for information on how to convert these strings to UTF8.

The following functions are available:

1. `MLuint16* MLConvertUTF8ToUTF16(const char* input)`

   Converts the given input char string (UTF8, terminated by 0) to UTF16, returns a newly allocated string that must be freed with `MLFree()`, returns `NULL` on error.

2. `MLuint32* MLConvertUTF8ToUTF32(const char* input)`

   Converts the given input char string (UTF8, terminated by 0) to UTF32, returns a newly allocated string that must be freed with `MLFree()`, returns `NULL` on error.

3. `char* MLConvertUTF16ToUTF8(const MLuint16* input)`

   Converts the given input wide string (UTF16, terminated by 0) to UTF8, returns a newly allocated wide string that must be freed with `MLFree()`, returns `NULL` on error.

4. `char* MLConvertUTF8ToLatin1(const char* input)`

   Converts the given UTF8 encoded string into a Latin1 string, converting all non-Latin1 chars to '?', the returned string must be freed with `MLFree()`, returns `NULL` on error.

5. `char* MLConvertUTF16ToLatin1(const MLuint16* input)`

   Converts the given UTF16 encoded wide string into a Latin1 string, converting all non-Latin1 chars to '?', the returned string must be freed with `MLFree()`, returns `NULL` on error.

6. `char* MLConvertLatin1ToUTF8(const char* input)`

   Converts the given Latin1 encoded string into a UTF8 string, the returned string must be freed with `MLFree()`, returns `NULL` on error.

7. `MLuint16* MLConvertLatin1ToUTF16(const char* input)`

   Converts the given Latin1 encoded string into a UTF16 wide string, the returned string must be freed with `MLFree()`, returns `NULL` on error.

Chapter 10. File System Support

Chapter Objectives

This chapter describes features and limitations of the ML with regard to managing platform-independent file (system) accesses with file names that contain international characters.

10.1. File System

Opening, reading, writing, manipulating and closing files is generally not platform-dependent; however, the way of coding file names to manage international characters is. The ML provides a set of functions to prevent file management from becoming difficult or from getting platform-dependent.

Generally, you should use these functions when you are not sure if the file names are unicoded or not.

Examples:

1. When you receive file names from string fields of ML modules, you need to handle them as UTF8-coded strings, and files need to be managed by using the functions provided by `mlFileSystem.h` .
2. When you have file names containing only ASCII characters, you can use the normal file functions offered by the system. However, it also possible to use the functions provided by `mlFileSystem.h` ; it makes your code more flexible concerning later changes to unicoded strings.

The following functions are available:

1. `FILE *MLfopen(const char *fileName, const char *mode)`

   Opens the file *`fileName`* with the access mode *`mode`*, returns a `FILE` pointer or `NULL` on failure. This method is equivalent to the `stdio fopen` implementation, see the `fopen` documentation for available mode flags ("r","w","a", etc.). In contrast to the original `fopen` function, this method accepts an UTF-8 encoded string and uses the unicode WIN32 API on Windows. On Linux, this method maps to `open` directly.

2. `int MLopen(const char *fileName, int openFlags, int pMode)`

   Opens the file with name *`fileName`* with the given *`openFlags`*, returns a file descriptor or -1 on error. This function is equivalent to the stdio `open` implementation, see the `open` documentation for available `open` flags. In contrast to the original `open` method, this method accepts an UTF-8 encoded string and uses the unicode WIN32 API on Windows. On Unix systems, this method maps to `open` directly. *`pMode`* specifies the access permissions of the opened file.

3. `int MLFileExists(const char *fileName)`

   Returns 1 if the file with the UTF8 coded name *`fileName`* exists, 0 otherwise.

4. `int MLFileIsReadable(const char *fileName)`

   Returns 1 if the file with the UTF8 coded name *`fileName`* exists and is readable, 0 otherwise.

5. `int MLFileIsWritable(const char *fileName)`

   Returns 1 if the file with the UTF8 coded name *`fileName`* exists and is writable, 0 otherwise.

6. `int MLFileWriteString(const char *fileName, const char* data)`

   Creates/overwrites the file with the UTF8 coded name *`fileName`* with the null terminated given data string *`data`* and returns 1 on success or 0 on error.

7. `int MLFileAppendStringData(const char *fileName, const char* data)`

Appends the null terminated data string *data* to the file with the UTF8 coded name *fileName* (creating a new file if it does not exist), returns 1 on success and 0 on error.

8. `char* MLFileReadAllAsString(const char *fileName)`

   Reads the complete file with UTF8 coded name *fileName* and returns its content as a null-terminated string or returns `NULL` on error. The returned memory needs to be deallocated by calling `MLFree()`.

9. `MLuint8* MLFileReadAllAsBinary(const char *fileName)`

   Reads the complete file with UTF8 coded name *fileName* and returns its content as binary data or returns `NULL` on error. The returned memory needs to be deallocated by calling `MLFree()`.

There are a number of additional functions available, see `mlFileSystem.h` for details:

1. `MLErrorCode MLfclose(FILE *file);`
2. `MLErrorCode MLremove(const char *fileName);`
3. `MLErrorCode MLrename(const char *oldName, const char *newName);`
4. `MLErrorCode MLclose(int fd);`
5. `MLErrorCode MLDeleteFile(const char *fileName);`
6. `char* MLGetNonExistingRandomFileName(const char *prefix);`
7. `MLErrorCode MLFileWriteStringData(const char *fileName, const char *str);`
8. `MLErrorCode MLFileWriteBinaryData(const char *fileName, const MLuint8 *data, unsigned int len);`
9. `MLErrorCode MLFileWriteBinaryDataAt(int fileDesc, MLint startPos, const MLuint8 *data, unsigned int len);`
10. `MLErrorCode MLFileAppendStringData(const char *fileName, const char *strData);`
11. `MLErrorCode MLFileAppendBinaryData(const char *fileName, const MLuint8 *data, unsigned int len);`
12. `MLErrorCode MLFileAppendBinaryDataWithDescriptor(int fileDesc, const MLuint8 *data, unsigned int len);`
13. `char* MLFileReadChunkAsString(const char *fileName, MLuint startPos, MLuint numBytes);`
14. `MLuint8* MLFileReadChunkAsBinary(const char *fileName, MLuint startPos, MLuint numBytes);`
15. `MLuint8* MLFileReadChunkAsBinaryFromDesc(int fileDesc, MLuint startPos, MLuint numBytes);`
16. `char* MLFileReadChunkAsStringFromDesc(int fileDesc, MLuint startPos, MLuint numBytes);`
17. `MLint MLFileGetSizeFromDescriptor(int fd);`
18. `MLint MLFileGetSizeFromName(const char *fileName);`
19. `MLint MLFileSetBytePos(int fd, MLint pos);`

See also Section 2.6.2, "`MLUtilities`" and Chapter 9, *Unicode Support* for more information on helper functions for the platform-independent implementation of file-system-related functions.

Appendix A. Basics about ML Programming and Projects

Objectives of This Appendix

This appendix will provide further information that is needed for ML programming even though it is not directly related to it.

A.1. Creating an ML Project by Using MeVisLab

The development version of MeVisLab fully supports easy creation of running ML projects for developers:

- Go to the **File** menu of the MeVisLab application and select **Run Module Wizard** . See corresponding chapter in the document **Getting Started.**
- Select `Inventor`, `Macro Module`, `ML` or `Load Setting` to create
  - C++ and project code for an Inventor node for 2D/3D visualization. See also the Open Inventor™ Toolmaker book which describes how you implement your own visualization node.
  - MDL (module description language) and Python code for a MeVisLab macro.
  - C++ code and project files for an ML image processing module.
  - or if you want to load project settings of a project you previously created.
- Follow the instructions and fill in parameters as shown in the module wizard.
- See the MeVisLab SDK (Software Development Kit) for additional information on project and software development in MeVisLab.

The wizard will open the directories where files have been created.

If you work with Visual C++™, you can open the project file <ProjectName>.vcxproj and compile the project.

If you work with Linux, you can compile the project with the created makefile.

If you work with Mac OS X, you can open and compile the project with the <ProjectName>.xcodeproj.

Then you can start MeVisLab and look for your module in the menu entries or the MeVisLab Module search.

Have a close look at the comments within the source code to get familiar with module programming. See Chapter 3, *Deriving Your Own Module from Module* for details.

Also have a look at the document **Getting Started** to learn the necessary steps in detail (and much more with regard to using MeVisLab).

A.2. Programming Examples

Some programming examples are available with the MeVisLab software development kit. Here is an overview of the most important ones.

- **mlAddExample**

  Startup example for ML module programming.

- **mlBitImageExample**

  This module demonstrates the `BitImage` class of the ML Tools project.

- **mlFieldExample**

An example module which simply creates most ML fields and adds them to a module interface. It also uses the new Vec8Field also derived in this library.

- **mlGlobalPagedImageExample**

  This module demonstrates how a `VirtualVolume` and/or a `TVirtualVolume` instance can be used to get a random read/write access to an input image during page-based processing and to demand driven image processing.

- **mlKernel3In2OutExample**

  Example class to demonstrate the implementation of a kernel-based algorithm with three inputs and two outputs in the ML.

- **mlKernelExample**

  Example class to demonstrate the implementation of a kernel-based algorithm in the ML.

- **mlMarkerListExample**

  Example module generating an equally spaced linear set of `XMarker` objects.

- **MLObjVolume**

  Example module to store and retrieve volume information in a hard-coded `ObjMgr` information cell. For details see the MeVisLab SDK.

- **mlProcessAllPagesExample**

  This is an example module to demonstrate how to process all pages of one or more (input) images.

- **mlSeparableKernelExample**

  Example class of the implementation of a kernel-based algorithm in the ML which implements separable kernel filtering.

- **SmallImageInterfaceExample1, SmallImageInterfaceExample2**

  Example modules to demonstrate the class `SmallImageInterface` which provides a very simplified image processing interface for educational use. See the MeVisLab SDK for details.

- **mlSparseImageExample**

  Defines an example module which uses a `VirtualVolume` as a sparse image.

- **MLTypeAddExample**

  Example class to demonstrate the integration of a new voxel data type in the ML.

A.3. Exporting Library Symbols

If you want to implement classes for your module or your module itself in such a way that other projects can also link their interfaces, you have to care about that in a particular way in order to make sure that these types, classes and symbols are available even on Windows systems. For doing this correctly, all symbols other libraries use must be *exported*; this is which is generally solved in ML projects by writing a macro in front of your (e.g.) class containing the export code. The macro is normally defined in the InitSystem.h file of your project where the platform-dependent stuff is implemented. The macro is usually something like this:

Example A.1. Exporting Library Symbols

```
//------ Solve platform dependent symbol exporting with macros --------------
#ifdef WIN32
#ifdef MLEXAMPLE_EXPORTS
// To make functions, classes and other symbols available
// on this DLL interfaces, they must be exported explicitly
// on win32 systems. We add simply MLEXAMPLE_EXPORT before
// them.
#define MLEXAMPLE_EXPORT __declspec(dllexport)
#else
//! When included by other libraries MLEXAMPLE_EXPORT is
//! compiled as import symbol.
#define MLEXAMPLE_EXPORT __declspec(dllimport)
#endif

#else
// Non windows systems:
//! Exporting library symbols is not used on non windows systems.
#define MLEXAMPLE_EXPORT
#endif
```

Class export is done with a code like this:

```
class MLEXAMPLE_EXPORT AddExample : public Module{
 ...
}
```

`MLEXAMPLE_EXPORTS` is defined in the project `CMakeLists.txt` file; on Windows platforms, all classes of the project, for example, will implement `__declspec(dllexport)` in front of the symbol so that it will be available on the library interface. Other projects that do not define `MLEXAMPLE_EXPORTS` will implement `__declspec(dllimport)` to mark the symbol as linked from another library.

Such symbols are not defined on non-Windows platforms, i.e., the `MLEXAMPLE_EXPORT` will have no effect on Linux, for example, because it is simply not compiled.

A.4. General Rules for ML Programming

There are many general rules an ML programmer should keep in mind:

- **Implement all ML related code in the namespace `ml`, even the source code (please use macros `ML_START_NAMESPACE` and `ML_END_NAMESPACE`)!**

  Many ambiguities can be avoided and global namespace pollution is reduced.

- **Use `mlDebug` macros, or - if you really cannot avoid it - use `std::cout` and `std::cerr`! Never use `(f)printf`, `cout` or `cerr` from global namespace!**

  Otherwise, the debug output cannot be controlled or disabled. `std::cout`, `std::cerr` and `mlDebug` statements can be redirected and disabled by the ML. Forgotten outputs of ML modules (hundreds of them exist) can be disabled and do not lead to output garbage in ML based applications. See module `RedirectStream` in project `MLStreamSupport` if you want to redirect `std::cout` or `std::cerr` to the ML error handler.

- **Never use `abort()`, `exit()` or other program terminating commands!**

  The ML cannot handle those terminations. Use dedicated ML macros for error handling as described in Section 5.2, “Handling Errors”.

- **Avoid global image processing algorithms**!

  Avoid them, even if they are sometimes faster or easier to implement. Always remember that ML-based applications often use hundreds of modules and that just some modules that work with global approaches can easily lock the entire memory so that is impossible for the ML-based applications to

run safely. The ML is dedicated to working safely with huge networks which, however, is only possible if programmers stick to the page-based image processing approach.

This is sometimes difficult, but see Chapter 4, *Image Processing Concepts* and Section 4.2.1, "Page-Based Concept", Section 4.2.4, "Kernel-Based Concept" and Section 4.3.2, "Sequential Image Processing Concept" for detailed information on the page-based, kernel-based, and sequential image processing concepts as well as Section 2.3.7, " `VirtualVolume` " for information on the `VirtualVolume` class to find an adequate algorithm approach that does not need too much memory.

- **Document your module well, test it and optimize it!**

  Although this rule should be self-evident, it is seldom observed which often leads to big problems:

  - Undocumented modules are useless (or to be even more precise: garbage) in a large module database. They are not usable (since nobody knows how to use them), they hinder database users from finding adequate modules by distracting the users and forcing them to spend time on checking the modules.
  - If such modules are used in macros (e.g., in MeVisLab), nobody has a chance to understand the macro or to find bugs.
  - Undocumented modules also tend to be buggy or untested which makes larger module networks unsafe and unstable.

  Always remember that especially MeVisLab applications often use hundreds of modules at once. This would not be possible with unstable modules.

  See Section A.5, "How to Document an ML Module" and Appendix B, *Optimizing Image Processing* for a description of an adequate module documentation and for information on how to optimize modules.

- **Use the ML functionality for error handling and memory allocation!**

  Only the ML can avoid exceeding memory usage and undesired application crashes. See Section 5.2, "Handling Errors", ConstructingAndDeletingObjects [111], HandlingExceptions [109], and Chapter 5, *Debugging and Error Handling* for more information.

- **Name field pointers in your modules as field pointers!**

  Name them, e.g., with the appendix "**Fld**" like "threshold**Fld**". That makes module code much easier to read.

- **Try to implement your algorithms for all 6 dimensions!**

  Programmers tend to forget that the ML supports fully 6D image processing and thus only implement 2D or 3D algorithms. Try to work high (6) dimensional algorithms; it is often easier than expected!

  However, there are also algorithms which become quite difficult; try to support 6D by bypassing higher image coordinates; so a 2D algorithm, for example, would be implemented in such a way that it filters all slices of a 3D (or higher dimensional) image independently. Thus, also images of a higher dimension can benefit from those algorithms even if it works only in 2D.

A.5. How to Document an ML Module

The following hints can help you to create a complete and useful documentation of your ML classes and modules:

- Use Doxygen/Dot for the documentation of the source code. Have a look at existing source code.
- Add author name, creation date, filename to the file header.

- Document the header file completely, that includes all members, methods, classes functions and types.
- Document functionality, usage, side effects and default values of interface components for classes, methods, functions, etc.
- The standard for Doxygen (http://www.stack.nl/~dimitri/doxygen/manual.html) with Graphviz/Dot (http://www.research.att.com/sw/tools/graphviz/) is the standard for the *header file and/or source code* documentation.
- When you use your modules in MeVisLab, your modules become really powerful. So make the ML module usable for MeVisLab. This includes the following steps:
  - Create an example network which demonstrates how your module works and insert this example network as a link into the .def file with which it can be called from MeVisLab.
  - Write a module help: choose Edit Help from the module's context menu to open MATE in mhelp mode.
  - Add keywords and cross references to the .def file. MeVisLab registers the module and other people can search for it in the MeVisLab module database. Use reasonable keywords so that people can find the modules in the MeVisLab databases.

These rules, of course, usually also apply to non-ML modules such as Open Inventor™ nodes or macros.

A.6. Updating from Older ML Versions

There is some stuff that should not be used or that is still supported by the ML but will be removed:

- Fields can still use external values as field values. This concept was implemented in the first ML version to make it easier to port modules from the old `ImgLab` application. However, using such references typically makes module programming more difficult than using the fields contents as values. The normal way to port such code is to remove the externally referenced values/members and to replace every occurrence of the value/member by `field->getValue()` calls or `field->setValue()` calls. Be careful when using `field->setValue()` calls because the setting of a value normally also notifies attached fields of that value change.
- You may sometimes find modules where field names start with capital letters or underscores, or where field names contain spaces, commas or other non-alphanumeric characters. Some of these field names are/were no error; however, they are not up to date anymore and can cause problems. Normally, a field should start with a lowercase letter and it should contain alphanumeric characters only. This makes module scripting (e.g., in applications such as MeVisLab) much easier and more reliable. Also, field names should be very similar to the names of the member variables managing those fields in the module code.
- Older modules often use a flag to suppress calls of the `handleNotification()` method in the module while field values are initialized in the constructor or changed elsewhere. The more reliable way is to use `handleNotificationOff()` and `handleNotificationOn()` which does not need additional code in `handleNotification()` or a flag member in the module. Also see Section 3.1.2, “Implementing the Constructor” for more information.
- Older modules often use `cout`, `cerr` or `std::cout`, `std::cerr`, `printf` or `fprintf` calls for debugging or error handling purposes. The same is also true for `exit()`, `abort()` or `assert()` statements. This is not desired in module programming because programmers tend to forget to remove such calls, i.e., they will forever print information to the output streams, or the error will not reach the central error handler of the ML. Replace those calls by `mlDebug` macros as described in Section 5.1, “Printing Debug Information” or by the corresponding error handling macros as described in Section 5.2, “Handling Errors”. The debug macros can also remain in the code and can be selectively enabled or disabled during runtime.

A.7. Version Control

The ML offers a version number and some features to check for version compatibility of related binaries. The project `MLUtilities` (see Section 2.6.2, "`MLUtilities`") contains the necessary file `mlVersion.h`. Also see Section A.7, "Version Control" for source code changes. Usually, there is no need for additional checks in your code, because the ML automatically checks for the correct version (e.g., when calling `MLInit()` of the ML C-API (see Section 6.3, "mlAPI.h") or when initializing a dynamically linked library with the `ML_INIT_LIBRARY` macro (see ML_INIT_LIBRARY [47])). It will print errors on version conflicts, but will not refuse operation when a version conflict occurs.

Important

The ML can check for correct versions when it is initialized and when dynamic linked libraries are linked on runtime. It, however, cannot check if different dynamic linked libraries are compatible between themselves.

The following macros are available on compile time for versioning:

1. `ML_MAJOR_VERSION`

   The major release number that indicates general and essential changes to the ML (which usually imply binary and header file incompatibilities).

2. `ML_MAJOR_CAPI_VERSION`

   Changes to this number indicate binary incompatibilities of the C-API of the ML which require a recompilation of applications using the ML via the C-API.

3. `ML_CPPAPI_VERSION`

   Changes to this number indicate binary incompatibilities of the C++ interface of the ML which require a recompilation of all classes using C++ ML symbols. Also, changes to this number sometimes indicate C++ header file incompatibilities. Note that the C++ API is also considered changed when the C-API has changed.

4. `ML_CAPI_REVISION`

   Changes to this number indicate a revision of the C-API of the ML which normally does not require a recompilation of applications using the ML via the C-API; this is typically caused by additional functionality in the C-API.

5. `ML_REVISION`

   Changes to this number indicate any revision of the ML which does not influence the binary compatibility (also docs, comments, installers); thus dependent classes do not need to be recompiled.

6. `ML_VERSION_STRING`

   The version string is put together by the above five strings, the individual strings are separated by ".". So, the version string would begin with:

   ML_MAJOR_VERSION.ML_MAJOR_CAPI_VERSION. (to be followed by the other three above strings).

The following functions are available for runtime version checks:

1. ```
   void MLGetVersion(int *majorVersion, int *majorCAPIVersion, int *verCPPAPI, int *revCAPI, int *rev, const char **vString);
   ```
   ```

Returns version information about the ML. It is legal to call it before the `MLInitializeUtils()` is called.

For all parameters, a NULL may be passed if that parameter is not needed.

- *`majorVersion`* Returns the compiled major release number specified by `ML_MAJOR_VERSION`.
- *`majorCAPIVersion`* Returns the compiled major C-API version number specified by `ML_MAJOR_CAPI_VERSION`.
- *`verCPPAPI`* Returns the compiled C++-API version number specified by `ML_CPPAPI_VERSION`.
- *`rev`* Returns the compiled ML revision number specified `ML_REVISION`.
- *`revCAPI`* Returns the compiled C-API revision number specified by `ML_CAPI_REVISION`.
- *`vString`* Returns a null terminated character string as "majorVersion.majorCAPIVersion.revCAPI.verCPPAPI.rev".

2. `int MLIsCAPILinkCompatible(int majorVersion, int majorCAPIVersion, int revCAPI);`

   Checks whether the ML API is link compatible. It is legal to call this function before `MLInitializeUtils()` is called. Normally, it is not necessary to call this function "manually" because the ML does these checks automatically when the ML is initialized or modules are loaded. A typical call looks like the following:

Example A.2. `MLIsCAPILinkCompatible`

```
if (!MLIsCAPILinkCompatible(ML_MAJOR_VERSION, ML_MAJOR_CAPI_VERSION, ML_CAPI_REVISION))
{
  handleErr();
}
```

   A non-zero value `(=true)` is returned if binary compatibility is given, and 0 if not.

   Parameters are

   - *`majorVersion`* The major release number of the ML (normally specified by `ML_MAJOR_VERSION`).
   - *`majorCAPIVersion`* The major C-API version number of the ML-API (normally specified by `ML_MAJOR_CAPI_VERSION`).
   - *`revCAPI`* The revision number of the C-API of the ML-API (normally specified by `ML_CAPI_REVISION`).

3. `int MLIsCPPAPILinkCompatible(int majorVersion, int majorCAPIVersion, int verCPPAPI, int revCAPI);`

   Checks whether the C++-API of the ML is link compatible. It is legal to call this function before `MLInitializeUtils()` is called.

   Normally, it is not necessary to call this function "manually" because the ML does these checks automatically when initializing the ML is initialized or modules are loaded. A typical call looks like the following:

Example A.3. `MLIsCPPAPILinkCompatible`

```
if (!MLIsCPPAPILinkCompatible(ML_MAJOR_VERSION,
                              ML_MAJOR_CAPI_VERSION,
                              ML_CPPAPI_VERSION,
                              ML_CAPI_REVISION))
{
  handleErr();
}
```

It returns a non-zero value (`=true`) if binary compatibility is given, and it returns 0 if binary compatibility is not given. Parameters are

- *`majorVersion`* The major release number of the ML (normally specified by `ML_MAJOR_VERSION`).
- *`majorCAPIVersion`* The major C-API version number of the ML-API (normally specified by `ML_MAJOR_CAPI_VERSION`).
- *`verCPPAPI`* The C++-API version number of the ML (normally specified by `ML_CPPAPI_VERSION`).
- *`revCAPI`* The revision number of the C-API of the ML-API (normally specified by `ML_CAPI_REVISION`).

Appendix B. Optimizing Image Processing

The following two sections discuss how to optimize image data flows in the ML and how to optimize module code.

B.1. Optimizing Module Code

- *Use a profiler* to analyze your module code.

  Very simple and unsuspicious code fragments can often cost a lot of time. Before optimizing irrelevant code find out where the time is actually spent.

- *Make sure that the time is really spent in your module.*

  Since an ML module usually does not work alone, it might happen that the time is spent in another module or in the ML internals. Loading images via networks, badly paged images, implicit data type conversions, changes to page extents, requests of big input subimages, etc. can require a lot of time which is not spent in your module.

- Make your image processing algorithm *inplace*.

  This is not a very powerful optimization, but it may result in a slight speed-up if you already have a fast algorithm.

- *Enable multithreading* for `calculateOutputSubImage()`.

  This enables the ML to call `calculateOutputSubImage()` in parallel. However, please be sure that your algorithm in `calculateOutputSubImage()` is really thread-safe to avoid nasty bugs.

- *Avoid position calculations* with 6D components.

  Often, a straightforward position calculation handles 6D positions. Methods which get vectors or a number of coordinates as parameters are usually expensive, because they require voxel address calculations in all or many dimensions which then can become quite inefficient in inner loops. Try to set a cursor (`setCursor*Pos()`) outside a loop and use the `moveTo*()` commands to move the cursor within the loop. This usually results in a simple and fast pointer-add operation because the compiler normally inlines that code.

- Try to *avoid changes of page extents* or be careful when selecting a new one.

  Changing page extents can result in a lot of expensive internal copying to compose input subimages for other modules. Try to leave the extent of pages unchanged; then the internal ML optimizations can recycle pages and page references optimally. When setting a new page extent, try to select one which is not too big or too small, and which has an extent of powers of two. If possible use the helper functions in `Module` to determine an optimal page extent.

- *Avoid inadequate page extents and inappropriate subimage requests.*

  Sometimes, page extents or image requests are not well suited, e.g., when you have images with page extent (128x128x1x1x1x1) and request a subimage from (10,10,0,0,0,0) to (10,10,50,0,0,0,0), a line of voxels perpendicular to all pages is requested. Hence, a large number of pages is processed, and only one pixel is copied from each page. This is of course expensive. Think about the (sub)image requests done in that pipeline and use adequate page extents when feeding an image into a module pipeline.

When a module network generally works e.g., slice-based with 2D viewers, 2D page extents are usually appropriate; when you work with 3D algorithms which usually work volume-based or when you are reformatting the image in different dimensions, 3D page extents might be useful, however, a 2D extent is also okay in most cases. To avoid administrative overhead, page extents should not be set too small.

Avoid page extents with dimensions that are higher than the dimension of the used image data, because otherwise the ML host has to manage unused data regions in pages.

- *Do not cast between data types* and do not try to change data types from module inputs to outputs if not really necessary.

  When you change data types, you are using cast operations that can become quite expensive on some systems, especially when casting floats to integers. This also inhibits inplace calculations and page recycling in the ML core.

- *Do not scale data if not really necessary*.

  When data is requested from the ML, this is often done by passing voxel value scaling information to the request so that the data is delivered in the right interval range. This can lead to expensive operations since implicit casting operations are often necessary then.

- *Try to implement your algorithm page-based*, i.e., select the optimal implementation approach for your algorithm.

  Algorithms which are not page-based (i.e., global image processing approaches) lock much memory; they often force the operating system to perfrom virtual memory swapping, they fill up the ML cache, and they often change page extents in a module pipeline, i.e., they do not work optimally with the optimized ML concept. When you need such algorithms, try to use approaches such as the `VirtualVolume` approach (Section 2.3.7, “ `VirtualVolume` ”) to merge global image processing with page-based approaches. Selecting the correct implementation approach can drastically speed up your algorithm. See Chapter 4, *Image Processing Concepts* for a detailed discussion of such approaches.

- *Request input subimages in "read only" mode.*

  The ML can pass pointers to cache pages directly as input subimages. That reduces memory allocations and copying in some cases. Note that this mode may not be available in some ML versions.

B.2. Optimizing Data Flow in Module Networks

- *Spend enough memory for the ML cache!*

  The ML image processing benefits strongly from sufficient cache memory. Usually, 30-50% of the main memory is a good value.

- *Reduce field notifications!*

  The more notifications are sent around through the network the more changes and calculations take place. Find out the really necessary field connections and changes and limit them to the minimum.

- *Avoid global image processing modules* or take them outside critical network branches!

  Global image processing modules (unfortunately, there are some in most networks) are often extremely expensive because they pull the entire image through the module pipeline and thus negate many advantages of page-based image processing. Solutions can be:

  - "Outsource" large images and expensive calculations. Calculate them once and store the results on disk. Then replace it by a `Load` module in the network. This, however, is often not possible, e.g., if module results change often.
  - Try to replace those module by other page-based solutions. Maybe other modules provide similar functionalities.
  - Move expensive calculations to less frequently used and changing parts of the data flow. Often - not always - the image data flow and the number of changes are higher near the output or viewer modules than directly after e.g., a `Load` module.
  - Reimplement the module and make it page-based, e.g., by using the `VirtualVolume` concept (see Section 2.3.7, “ `VirtualVolume` ”). Although this is sometimes difficult and a page-based approach may be slower considering the local processing in the module, the page-based image flow is not interrupted. This can result in a significant performance boost since data flow can be reduced.

- *Avoid or reduce unnecessary changes of image properties* (especially page extents, data types, image extents, etc.) in the image data flow!

  Changing image properties from one module to another usually requires expensive casting and/or copying of the image data or also a recomposition of pages.

- *Set number of permitted threads to the number of CPUs in your system!*

  Multithreading (parallelization) currently works optimally if the number of permitted threads in the ML matches the number of CPUs in your system.

- *Increase performance by reducing the memory optimization mode!*

  If there is enough memory, you can usually increase performance by reducing the memory optimization mode to lower numbers or even to zero. Hence more intermediate results are saved in the cache and the number of recalculations is reduced.

- *Consider the image format, compression and source when loading data from files!*

  Loading data can become slow when the file needs to be transferred via network connections or when the file format is compressed. Try to load files from local disks and/or store them uncompressed if you have enough disk space. Compressing files does not save memory when the image is compressed with ML modules. If the file format supports paging, store the file with a page extent adequate for image processing.

- *Increment the memory optimization mode to optimize memory usage!*

  If your network suffers from a lack of memory, increment the memory optimization mode to optimize memory usage; more pages are recalculated and less pages are buffered in the cache. This, however, usually reduces image processing speed.

- *Use release versions of the ML and MeVisLab!*

  When you develop your own software with the ML or with MeVisLab, you may probably work in debug mode and non-optimized code. Compiling release-mode code with optimizations may drastically speed up your applications.

- *Disable (symbol controlled) debugging!*

  Working in debug mode with symbol-controlled debugging may degrade performance during operation, because information is printed to the output. Disable symbol-controlled debugging or use release version code which automatically does not contain such code.

Appendix C. Handling Memory Problems

The ML is designed to work with large images and with many modules that work on these images. This, however, does not mean that working with large images and many modules is no problem anymore. It is still possible to run into memory problems that the ML cannot avoid automatically. In most cases, these problems can be solved by reconfiguring some settings.

1. *Check whether you machine has enough main memory!*

   Problem: The computer does not have sufficient physical memory.

   Possible Solution: Theoretically, pure ML programs could work with only a few MB of main memory. In many cases, however, the processed images and the applications using the ML programs will strongly benefit from more memory. 256 MB is considered to be a reasonable minimum memory size; assign as much memory as possible to achieve optimal results. Working with a memory of less than 256 MB might be possible but will often lead to slow performance and will also require the ML and the images to be configured appropriately. The following items might help you to work with less than 256 MB, but they do not guarantee success.

2. *Check whether other applications use too much memory!*

   Problem: Other applications use too much memory.

   Possible solution: Terminate other applications running on your system, especially those which use much memory. To find out which application uses much memory, check the Task Manager (Windows systems) or use "top" (Linux systems).

   If you do not want or cannot terminate those applications, it might help to make these applications sleep or to set them to an inactive state; the system can swap the memory that these applications use into the virtual memory and your ML process can use the physical memory more efficiently. That, however, might require you to increase the virtual memory size of your system.

3. *Set an adequate ML cache size!*

   Problem: The ML uses the cache to reduce the number of required page recalculations. If the user spends too much cache memory, the ML will try to use it, even if the required physical memory is not available which might lead to a memory allocation failure. This problem often occurs after a certain working time, because the cache needs some time to be filled with calculated data.

   Possible solution: Take the physical memory size of your computer, subtract the memory other applications and the system require (to find out these values you may want to use the "Task Manager" on Windows systems, or "top" on Linux systems), and also subtract the memory that the application using the ML needs (in most cases MeVisLab which works fine with about 256 MB). The remaining memory size should be the maximum limit for your cache size - better use less when you are not absolutely sure. A cache size that is too small can degrade image processing performance, but normally does not lead to memory problems since the ML will always use the minimum memory requirements for image processing, even if it exceeds the cache size.

   ### Note

   MeVisLab 2.0 and newer versions will not have an ML cache anymore but a global cache for ML and other libraries such as GVR or MeVisAP. The cache limit should be set to the size of the available free memory there.

4. *Avoid global image processing* modules or take them outside critical network branches!

Problem: For different reasons, some ML modules request or lock so much memory that there is not enough memory for the system and the ML. This is often caused by inadequate or lazy algorithm programming, performance requirements or bad parameter settings of the ML or some modules.

Possible Solutions:

- Try to replace those modules by other page-based solutions or - if the modules have such a parameter - select a page-based algorithms setting. There might be other modules which perform similar or the same tasks.
- Outsource large images and memory-expensive calculations. Calculate them once and store the results on disk. Then replace them by a `Load` module in the network. This, however, is often not possible, e.g., when module results change often.
- Reimplement the module and make it page-based, e.g., by using the `VirtualVolume` concept (see Section 2.3.7, “ `VirtualVolume` ”) or by using more efficient or packed data structures such as the `BitImage` concept (see e.g., Section 4.4.3, “`BitImage` Concept” for details) to manage flag images.

5. *The processed images are too large*

   Problem: The ML cannot process images with inappropriate page and image extents; e.g., extents with dimensions of more than 2^{31} or pages with more than 512K voxels.

   Possible Solution: The ML can process images with up to 2^{44} voxels, even on 32 bit systems. However, the extent in each dimension should not exceed 2^{31} and the number of pages per image is limited to 2^{23}. Hence, try to avoid extreme extents in any dimension and too small and also too large pages.

6. *The page extent is too large*

   Problem: One or more modules in the network set a page extent which is too large (e.g., sometimes modules use image extent for page extent). This leads to a degeneration of the image processing process in the network. As a result, paging, caching, multithreading and effective memory usage do not work appropriately anymore.

   Possible Solution:

   - If the loaded input image has already an inadequate page extent (e.g., the page extent is identical to the image extent), try to load the module, set the page extent to a smaller value, e.g., with the `ImagePropertyConvert` module, and save the image with this new page exent under a new filename. Use that new image instead of the original one.
   - Try to find those modules in the network and the reasons why they specify an inappropriate page extent. If possible, reconfigure or replace these modules in such a way that large page extents do not occur anymore.
   - If the modules cannot be reconfigured or replaced you may want to revise the module in such a way that it does not set these page extents anymore; that might be sensible because setting such an page extent is "bad module behavior" and may cause other users to have the same problem in the future.

7. *The process runs continuously out of memory after long usage*

   Problem: Although it should not happen in well-programmed code: module networks often include a large number of stable and new modules and sometimes some of these modules have memory leaks that result in memory problems after longer operation.

   Possible Solutions:

- Check the memory your process uses (use e.g., "Task Manager" on Windows or "top" on Linux) and check whether this behavior remains when you have temporarily deactivated some network components or some functionality. Thus you can isolate the module(s) that cause such a problem.

Note

Be aware that the ML also caches memory. To distinguish memory leaks from cached image fragments, use the `Clear Image Cache` in the menu `Extras` of MeVisLab or the `Clear Cache` feature of the `CoreControl` module.

- Use software tools or libraries which allow for checking for memory leaks. It might be helpful to check all modules used with the module tester of the MeVisLab application (if you work with MeVisLab); this could be the fastest way to detect memory leaks. You can also use the `Tester` module of the project `MLDiagnosis`.

8. *If all the above measures do not help*

   The measures might reduce memory usage and could be helpful; they, however, do not solve the actual problem:

   - Try to work on downscaled images or on image fragments.
   - Use 2D slice viewing instead of 3D volume rendering.
   - Do not use the `MemCache` or other memory caching modules.
   - Increase the memory optimization mode in the `CoreControl` module; this reduces the cache load at the expense of computing performance. Note: This mode is not available anymore in MeVisLab 2.0 or newer versions.
   - Simplify your module network and/or use smaller subnetworks to process images step by step and not at once.
   - Disable multithreading, because it temporarily uses more memory than single threading.
   - Increase the virtual memory size of your computer. This could increase reliability of the process but also may degrade performance if it is used too much.
   - Think about adding more memory to your computer, if possible.
   - Migrate to a 64 bit MeVisLab/ML version if you have not done so yet, and/or buy more memory.

Appendix D. Messages and Errors

Error messages and other messages are usually sent to the ML `ErrorOuput` class, where they are sent to all registered handlers which need to handle them (see Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs”). Not only messages and errors from modules or from the ML are sent to those handlers but also messages from other libraries or applications. In MeVisLab, for example, MeVisLab itself and the Open Inventor™ library redirect their outputs to the ML ErrorOutput. Hence, there is a large variety of messages. The following list only describes the currently known predefined messages and errors.

D.1. ML Error Codes

The following list explains each predefined ML error code. Note that other error codes may appear which are registered by applications or modules for advanced error handling. Refer to the corresponding documentation in such cases.

1. (MLErrorCode 0) ML_RESULT_OK - "Ok"

   No error. Everything seems to be okay.

2. (MLErrorCode 1) ML_UNKNOWN_EXCEPTION - "Unknown exception occurred"

   An unknown exception has been detected and caught. This usually means that something - for an unknown reason - went absolutely wrong and which should normally result in a program crash which is detected by the ML or a module. Look for previous errors, they may give more precise information. Try to reproduce this error and report it to the developer.

3. (MLErrorCode 2) ML_NO_MEMORY - "Memory allocation failed"

   The system does not have enough memory to perform the desired operation. Try to reduce application data and/or complexity, try to replace modules which load an entire image into the memory, terminate other applications running at the same time, buy more memory, etc.

4. (MLErrorCode 3) ML_DISCONNECTED_GRAPH - "Operator graph disconnected"

   The module/operator graph is obviously disconnected but expected to be connected for this operation.

5. (MLErrorCode 4) ML_CYCLIC_GRAPH - "Operator graph has cycle"

   The module/operator graph is connected cyclically. The ML cannot handle this. Search for the cyclic connections and remove them. Normally, this error should not occur.

6. (MLErrorCode 5) ML_BAD_OPERATOR_POINTER - "Bad operator pointer"

   A NULL, an invalid or a wrong module/operator pointer has been passed to an algorithm.

7. (MLErrorCode 6) ML_BAD_OPERATOR_OUTPUT_INDEX - "Bad index of output image"

   A bad output index of a module/operator has been specified.

8. (MLErrorCode 7) ML_BAD_FIELD - "Bad field pointer or name"

   A NULL, an invalid or badly/wrongly typed or named field has been passed to an algorithm.

9. (MLErrorCode 8) ML_IMAGE_DATA_CALCULATION_FAILED - "Calculation of image data failed"

   The requested image data could not be calculated. There is a variety of possible reasons. Look for previous errors, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer.

10. (MLErrorCode 9) ML_NO_IMAGE_INPUT_EXTENSION - "Calculation of required image input extension failed"

    Currently not used.

11. (MLErrorCode 10) ML_NO_IMAGE_PROPS - "Calculation of image properties failed"

    The calculation of image properties failed. There is a variety of possible reasons. Normally, this is a return code of functions accessing modules which cannot calculate a valid output image (this is often a legal state). If this is reported as an error or even a fatal error, look for previous errors, they may give more precise information. Report it to the developer if it seems to be a technical problem and not the report of a normal output state of a module.

12. (MLErrorCode 11) ML_BAD_OPERATOR_INPUT_INDEX - "Index to operator input is invalid"

    A bad input index of a module/operator has been specified.

13. (MLErrorCode 12) ML_BAD_INPUT_IMAGE_POINTER - "Pointer to input image is invalid"

    A NULL, an invalid or badly/wrong sized/typed image pointer has been passed to an algorithm. If no previous errors occurred, it might indicate a programming error or missing checks for invalid input connections, bad in/output indices, etc.

14. (MLErrorCode 13) ML_BAD_DATA_TYPE - "Bad data type"

    A wrong or unexpected data type has been passed to an algorithm. This is often a programming error. There is a variety of possible reasons. Look for previous errors, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer.

15. (MLErrorCode 14) ML_PROGRAMMING_ERROR - "Programming error"

    A situation occurred which should not appear. There is a variety of possible reasons; typically, it is a programming error in a module. Look for previous errors, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer.

16. (MLErrorCode 15) ML_EMPTY_MESSAGE - "<No Error Message>"

    The following error message describes more precisely what has happened. If not, a non-registered error occurred which is only known in the module where the error appeared. Have a look at the documentation of the module that produced the error. This code might also be passed with messages which are of another type, e.g. with debug information or user information.

17. (MLErrorCode 16) ML_PAGE_CALCULATION_ERROR_IN_MODULE - "Page calculation error in module"

    An image page could not be calculated. There is a variety of possible reasons. Often, this is a programming error in a module, but it can also be a result of an interrupted image processing in a module pipeline. Look for previous errors, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer if the error is not the result of a controlled interruption.

18. (MLErrorCode 17) ML_PROPERTY_CALCULATION_ERROR_IN_MODULE - "Property calculation error in module"

    Image properties could not be calculated correctly. There is a variety of possible reasons. Often, this is a programming error within a module. Look for previous errors, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer.

19. (MLErrorCode 18) ML_INBOX_CALCULATION_ERROR_IN_MODULE - "Inbox calculation error in module"

The input image region required the calculation of an image page which, however, could not be calculated correctly. This is often a programming error within a module leading to a crash and MLErrorCodes which was detected by the ML. There is a variety of possible reasons. Look for previous errors, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer.

20. (MLErrorCode 19) ML_BAD_PARAMETER - "Bad parameter"

    A bad/invalid parameter (or even an inappropriate image) has been passed to a module or an algorithm. This usually means that an invalid or inappropriate parameter has been passed to an algorithm, that the developer has forgotten to implement a certain case or that a parameter is out of range. Read the subsequent error information on how to handle this error.

21. (MLErrorCode 20) ML_CALCULATION_ERROR - "Calculation error"

    This is an error code used in some cases when the error is not very specific. There is a variety of possible reasons. Often, a programming error in a module caused a crash which was detected and handled by the ML. Some diagnostic modules also use this error code, e.g., to notify of an error about invalid calculation results, for example. Look for previous errors and additional error information shown with this error, they may give more precise information. Try to reproduce this error and its circumstances and report them to the developer.

22. (MLErrorCode 21) ML_BAD_DIMENSION - "Bad image dimension"

    The image or data structure has a wrong extent or wrong dimensions.

23. (MLErrorCode 22) ML_RECURSION_ERROR - "Invalid recursion"

    An invalid recursion occurred. When detected, it is usually broken to avoid subsequent crashes, but it usually also returns invalid results which also might lead to further errors. Often, this error occurs when ML image data is converted/rendered into a 3D OpenGL or Inventor Scene (e.g., by volume or iso surface rendering) which again is converted to an ML image (e.g., by snapshot or rasterization modules). This leads to invalid reentrances into the ML during image processing which are broken and commented by this error.

    A solution might be to reconfigure your module network so that module connections (image or node) from Inventor to ML and again to an Inventor node do not exist anymore. It also could help to complete an ML image calculation depending on Inventor node(s) before another Inventor/Viewer module requests image data from that ML module.

24. (MLErrorCode 23) ML_LIBRARY_LOAD_ERROR - "Library load/init failed."

    Loading or initialization of an ML module library failed. The shared library file may not exist at the searched place, a path to the libraries may be wrong, the library may not be up to date, symbols in the library interface may be missing or the library is of another or outdated version. The installation could be incomplete or damaged.

25. (MLErrorCode 24) ML_FILE_IO_ERROR - "File IO error" Opening, closing, reading, writing or searching of any file failed.

    There is a variety of possible reasons: A wrong file path may have been specified, other applications may use the file, file permission may be wrong, disk space may be not sufficient, etc.

26. (MLErrorCode 25) ML_AFTER_EFFECT - "Error due to previous error(s)"

    This is a typical error that occurs when another previous error has left an incomplete or undefined state. Look for previous errors, they may give more precise information.

27. (MLErrorCode 26) ML_BAD_INDEX - "Bad index"

The index given to the algorithm is out of range. Sometimes, this is a programming error or due to an (user) interface that has been sloppily implemented and passes invalid user inputs.

28. (MLErrorCode 27) ML_OUT_OF_RANGE - "Out of Range"

    A coordinate or value is out of range, often a voxel address which is outside of an image. Often, this is a programming error or caused by using an image with an invalid content.

29. (MLErrorCode 28) ML_MISSING_VOXEL_TYPE_OPERATIONS - "Missing voxel type operations"

    A voxel data type does not implement the required arithmetic operations. Often, this is a programming error. This error also indicates that a module does not support calculations on the connected input voxel type.

30. (MLErrorCode 29) ML_BAD_FIELD_TYPE - "Bad field type"

    The passed parameter is not derived from the class field or is not of the expected field type. This can be a programming error.

31. (MLErrorCode 30) ML_BAD_FIELD_POINTER_OR_NO_MEMORY - "Bad field pointer or memory allocation failed"

    The passed parameter is not of an expected (field) type or the allocation of memory failed.

32. (MLErrorCode 31) ML_FIELD_CREATION_ERROR_OR_NO_MEMORY - "Field creation error or memory allocation failed"

    A field could not be created (e.g., because the field type is still not registered in the runtime type system or the corresponding shared library is still not loaded) or the field creation failed due to lack of memory.

33. (MLErrorCode 32) ML_TYPE_INITIALIZATION_ERROR - "Type initialization error"

    A (runtime or voxel data) type could not be initialized correctly.

34. (MLErrorCode 33) ML_CONSTRUCTOR_EXCEPTION - "Exception in new"

    Creating an object failed due to a programming error in a constructor or due to lack of memory.

35. (MLErrorCode 34) ML_DESTRUCTOR_EXCEPTION - "Exception in delete"

    The destruction of a C++ object failed, e.g., due to a programming error or because it was destroyed by other buggy code.

36. (MLErrorCode 35) ML_TABLE_FULL - "Table full"

    A table is full and nothing can be inserted anymore.

37. (MLErrorCode 36) ML_EXTERNAL_ERROR - "Error from external library or application"

    Error messages from other libraries are delivered with this error code if more specific error information from the external library is not available.

38. (MLErrorCode 37) ML_BAD_BASE_FIELD - "Bad base field type"

    The (runtime) type of a `Base` field is not the expected one, the `Base` field pointer is invalid (NULL) or it is not (derived from) a `Base` field.

39. (MLErrorCode 38) ML_BAD_BASE_FIELD_CONTENT - "Bad content in base field"

    The pointer content of the `Base` field is invalid, i.e., it should not be NULL or it does not point to an object derived from `Base`.

40. (MLErrorCode 39) ML_TYPE_NOT_REGISTERED - "Required type not registered"

    The required or used type is (still) not registered. You probably forgot to call "YourClassName::initClass" in your initialization file, or there is a missing linked library which contains the type but which has not been loaded yet. Maybe a library dependency has been forgotten in the project (make) file or types are initialized in the wrong order in a library init file.

41. (MLErrorCode 40) ML_LIBRARY_INIT_ERROR - "Library init failed"

    The initialization code of a library failed. This is a typical error when the ML, an application or a linked library has detected an initialization problem. This can, for example, be due to an invalid version number (i.e., a binary incompatibility), forgotten recompilations of self-defined libraries, paths to (outdated) linked libraries, etc. or the usage of incompatible library and application installers.

42. (MLErrorCode 41) ML_BAD_POINTER_OR_0 - "Bad pointer or 0"

    A pointer is NULL or a value is NULL or 0 where it should not be. This sometimes indicates a memory allocation error, a programming error, a forgotten NULL pointer check at function entries or also bad function results or objects which have not been found.

43. (MLErrorCode 42) ML_BAD_STATE - "Bad state"

    The current state of an object is not appropriate for an operation. Maybe it is not initialized or in a valid but inadequate state. This also might indicate that the program ran into an undefined state which should not be possible.

44. (MLErrorCode 43) ML_TOO_MANY_PUSHES_OR_ADDS - "Too Many Pushes Or Adds"

    Too many elements were pushed or added onto a stack, array or another container type.

45. (MLErrorCode 44) ML_TOO_MANY_POPS_OR_REMOVES - "Too Many Pops Or Removes"

    Too many elements were removed from a stack, an array or another container type.

46. (MLErrorCode 45) ML_STACK_TABLE_OR_BUFFER_EMPTY - "Stack Table Or Buffer Empty"

    The access to a table, stack or container or its elements failed, because it is empty.

47. (MLErrorCode 46) ML_STACK_TABLE_OR_BUFFER_NOT_EMPTY - "Stack Table Or Buffer Not Empty"

    A table, stack, or another container was expected to be empty, but it is not.

48. (MLErrorCode 47) ML_ELEMENT_NOT_FOUND - "Element Not Found"

    An expected entry or element was not found.

49. (MLErrorCode 48) ML_ - "InvalidFileName"

    The specified file name is not valid, for example, because it is empty or because it contains invalid characters or path specifications or simply because it does not specify a correct file.

50. (MLErrorCode 49) ML_INVALID_FILE_DESCRIPTOR - "InvalidFileDescriptor"

    The descriptor used to manage a file is invalid or denotes a closed file.

51. (MLErrorCode 50) ML_FILE_NOT_OPEN - "FileNotOpen"

    The specified file is not open.

52. (MLErrorCode 51) ML_NO_OR_INVALID_PERMISSIONS - "NoOrInvalidPermissions"

The operation cannot or could not be executed because the user or the process does not have appropriate permissions or the permissions of the object to manipulate are not set correctly.

53. (MLErrorCode 52) ML_DISK_OR_RESSOURCE_FULL - "DiskOrResourceFull"

    There are not enough resources left to execute the desired operation. This typically indicates that the disk is full or that there is not sufficient memory for this operation.

54. (MLErrorCode 53) ML_FILE_OR_DATA_STRUCTURE_CORRUPTED - "FileOrDataStructureCorrupted"

    The content of a file or another data structure is not organized as expected by the program. This may indicate a broken file, an overwritten data structure or sometimes a newer and still unknown version of a file. It could also indicate a file or data structure created by another application which uses the same named types or files.

55. (MLErrorCode 54) ML_INVALID_VERSION - "InvalidVersion"

    The version of a data structure of file is invalid; maybe the version is newer than expected. An update of the software could help.

56. (MLErrorCode 55) ML_UNKNOWN_OR_INVALID_COMPRESSION_SCHEME - "UnknownOrInvalidCompressionScheme"

    The compression scheme is invalid, too old, too new or not known on your system. A compression scheme could have been used on another system to store a file which cannot be loaded on the local system, because the (de)compressor is not known on the local system. It could also indicate a corrupted data structure or file, or even a library that is missing or has not been installed..

57. (MLErrorCode 56) ML_TYPE_ALREADY_REGISTERED - "TypeAlreadyRegistered"

    This error occurs on an attempt to register a type whose name is already registered. This might, for example, happen when the system detects backup copies of modules or libraries and tries to load them, when a type initialization is called more than once, when older library paths are besides the current ones or when two developers independently developed types or classes with the same name.

58. (MLErrorCode 57) ML_TYPE_IS_ABSTRACT - "TypeIsAbstract"

    The runtime type to be used is abstract and cannot be used (an object of that type, for example, cannot be created then).

59. (MLErrorCode 58) ML_TYPE_NOT_DERIVED_FROM_EXPECTED_PARENT_CLASS - "TypeNotDerivedFromExpectedParentClass"

    The used class type is not of the expected type and/or is not derived from the expected parent/base class.

60. (MLErrorCode 59) ML_OPERATION_INTERRUPTED - "OperationInterrupted"

    The operation was interrupted, either by a user or another signal.

61. (MLErrorCode 60) ML_BAD_PAGE_ID - "BadPageId"

    This error comments the attempt to use an identifier or index to an (image) page which does not exist or which is out of range.

62. (MLErrorCode 61) ML_OUT_OF_RESSOURCES - "OutOfRessources"

    There are not enough resources to execute the desired operation. This might, for example, happen when the maximum number of open files, processes, threads, etc. is exceeded, or when the operating system does not have sufficient memory for the desired operation.

63. (MLErrorCode 62) ML_OBJECT_OR_FILE_EXISTS - "ObjectOrFileExists"

    The object or file to be created already exists.

64. (MLErrorCode 63) ML_OBJECT_OR_FILE_DOES_NOT_EXIST - "ObjectOrFileDoesNotExist"

    The expected object or file does not exist or is not found.

65. (MLErrorCode 64) ML_DEADLOCK_WOULD_OCCURR - "DeadlockWouldOccurr"

    The operation cannot be executed because it would lead to a deadlock.

66. (MLErrorCode 65) ML_COULD_NOT_OPEN_FILE - "CouldNotOpenFile"

    The file could not be opened because, for example, the permissions are not sufficient, resources for opening are not available, the file could not be found, or the file is already open.

Appendix E. Improving Quality of ML-Based Software

The ML and its modules are often used in contexts where robustness and reliability are of crucial concern. This is especially true for MeVisLab which uses the ML for image processing in medical applications to a large extent. Therefore remember the following aspects when you develop software based on the ML:

- General Software Quality

  See Section A.4, “General Rules for ML Programming” for strategies on how to improve software quality, and on how to simplify maintenance of source codes, modules and ML-based applications.

- Logging

  All tracing information, messages, warnings, and errors are sent to the ML error manager. Application developers can install a callback functionality there and redirect all this information to (application) specific output channels. See Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs” and Section 5.3, “Registering Error Handlers” for details and how an application and the ML error manager can be configured to receive all messages from the ML.

- Debugging Support

  See Chapter 5, *Debugging and Error Handling* with subsections Section 5.1, “Printing Debug Information”, and Section 5.4, “The Class `ErrorOutput` and Configuring Message Outputs” for information on debugging.

- Robustness of Source Codes, and Error Management and Detection

  See Section 5.5, “Tracing, Exception Handling and Checked Object Construction/Destruction”, Section 5.2, “Handling Errors”, and Appendix D, *Messages and Errors* for information on crash-safe function development, safe resource allocation and releasing, available error codes, and their meaning. See Section A.7, “Version Control” for information on how checks for correct ML versions can be implemented.

- Memory and Performance Risks

  Further potential problems in applications are out-of-memory situations and too slow or even hanging program executions. See Appendix C, *Handling Memory Problems* for strategies on how to configure the application for safe and limited memory consumption. See Appendix B, *Optimizing Image Processing* and the subsections Section B.1, “Optimizing Module Code” and Section B.2, “Optimizing Data Flow in Module Networks” for details on performance optimizations in module code and in module networks.

- Documentation

  The module data base, its use and its maintenance require certain documentation standards on source code level and on user level. See Section A.5, “How to Document an ML Module” for information on the recommended documentation.

Glossary

`Module` — The base class for all image processing modules. By overloading its methods and changing the class configuration, all desired image processing algorithms can be implemented.

`MLMemoryManager` — The library which manages buffers in memory that store intermediate results of (image) calculations which potentially will be reused. By reusing them, time for recalculating them is saved.

`Field` — A C++ class or object which usually encapsulates a data value, e.g., an integer or a vector. Fields can be observed so that the observer is notified when the field is changed, and fields can be connected among each other so that values changes are automatically propagated to other fields. Fields have a type and their values can be set or retrieved as a string or as a typed value. Thus, values between different typed fields can also be propagated as string values.

`Kernel` — A usually rectangular matrix or array of values or only a region which is placed onto a image voxel. A certain region around a voxel is specified and it is easy to calculate a new voxel value from that area. Often, a kernel is moved over all voxels of an image to filter it. Typical kernel operations on images are smoothing, sharpening, dilation, erosion, rank filters and many more.

DICOM — DICOM = ***Di****gital Imaging and* ***Co****mmunications in* **M** *edicine standard*. Standard communication protocol and file format for medical image data and information.

Doxygen — Doxygen is a documentation system for C++, C, Java, Objective-C, IDL (Corba and Microsoft® flavors) and to some extent PHP, C# and D. It is used to document the source of MeVisLab, the ML and ML modules. See http://www.stack.nl/~dimitri/doxygen/index.html for more information on this free software.

Exceptions (Catching) — A way to handle (un)intentional errors or undesired/special states in programs. Often used to detect and/or handle errors in programs.

FieldContainer — A C++ class used especially in `Module` objects to store and manage a list of fields containing parameters for ML modules.

MeVisLab — The *Image Laboratory*, a toolkit for rapid prototyping and development of applications. It uses the ML and Open Inventor™ a lot and offers specialized features for medical imaging.

Inplace Calculation — Usually an algorithm has input (image) data in one or more buffers and calculates a result written in an output buffer. This requires at least two buffers. Some algorithms can write the result directly into the input buffer(s) which is also the output buffer at the same time which then work or calculate *inplace*. Hence the creation/initialization and destruction of a buffer is spared which usually results in better performance.

Internationalization — Internationalization is the capability of software to be used in systems with different languages. This e.g., requires translated texts and unicode support (see Chapter 9, *Unicode Support*).

ITK™ — `The Insight Segmentation and Registration Toolkit ™`. A large and well known open-source image processing library which has

	been wrapped in many parts for MeVisLab to work fine with other ML modules. See www.itk.org and www.mevislab.de for details.
Lazy Evaluation	Information is only processed/evaluated by the ML when it is really needed or requested. Otherwise, the ML is lazy. This is similar to *Processing On Demand* or the *Pull Model*.
ML	The MeVis Image (Processing) Library.
Multithreading	The possibility to execute program code in parallel, e.g., of CPUs which can result in performance gains. Thus, two pages of an image, for example, can be processed in parallel by two CPUs which is faster than processing the pages sequentially. Such a program code, however, must fulfill some requirements so that programming with multithreading may become difficult.
`Node`	The `SoNode` class is a base class inherited by many Open Inventor™ classes to implement 3D objects, their properties or behavior, like 3D text, cubes, cones, transformations, colors, textures, etc. They usually can be composed to a 3D scene graph to build a 3D visualization.
Open Inventor™	An object-oriented 3D toolkit, a library of objects and methods used for interactive 3D graphics.
Page	An image sub-region of predefined extent; an (paged) image can usually be composed of a set of non-overlapping rectangular pages of identical extent. Pages may reach outside an image. A page is also a tile (or subimage), but tiles (subimages) are not necessarily pages.
Page-Based Image Processing	Processing an image not as a whole but in fractions, where only those fractions of the image are calculated which are really needed to achieve the result.
Processing On Demand	Results are only calculated when they are really needed; so a display needs to request the data it wants to show; i.e nothing is calculated without that request. This is similar to *Lazy Evaluation* or *Pull Model*.
Pull Model	Information is pulled (from a Viewer, for example) by other modules before the information is shown in a display; this is similar to *Processing On Demand* or *Lazy Evaluation*.
Runtime Type System	Database that stores information about many or all important class types of the library. It is also used to create instances from classes specified by a string name, to retrieve inheritance information and the name of the dynamic linked library where it comes from.
`SubImage`	A (usually rectangular) sub-region of an image; synonym for "tile".
Tile	A (usually rectangular) sub-region of an image; synonym for "subimage".
Unicode	Unicodes are used to provide string encodings with international characters. They are needed when language-specific characters or e.g. Chinese symbols are to be handled in strings.
`VirtualVolume`	Permits access to a paged image as if it was a global image. Only used image pages are mapped into memory, all other areas are not. Hence, an image of a potentially unlimited size can be handled using only a minimum amount of memory.
Voxel	The entity an image is composed from. Usually, one number such as an integer or a floating point number, but sometime also a structure

	containing several entities. The term "voxel" is made up from the words **Vo**lume and pi**xel** (***Pic****ture* ***El****ement*).
VTK™	`The Visualization Toolkit` ™. A large and well known open-source visualization library which has been wrapped in many parts to work also in MeVisLab. See http://www.vtk.org and https://www.mevislab.de for details.
Wizard	A *wizard* (more specifically called *Module Wizard* in this document) is a tool supporting a developer to create ML modules and the context needed for compiling and integrating the ML modules into an application. A module wizard is provided by MeVisLab which uses ML modules a lot. See Appendix A, *Basics about ML Programming and Projects* for details.