|  |  |
| --- | --- |
| Universität Stuttgart  ITO Institut für Technische Optik  Pfaffenwaldring 9  70569 Stuttgart  www.uni-stuttgart.de/ito |  |

itom-Unittest

12.10.2012

**DataObject**

**Inhalt**

[How to read this document 2](#_Toc337826474)

[Class Range 2](#_Toc337826475)

[Test 1 2](#_Toc337826476)

[Class DataObjectTagType 2](#_Toc337826477)

[Test 1 2](#_Toc337826478)

[Test 2 2](#_Toc337826479)

[Test 3 3](#_Toc337826480)

[General methods in dataobj.h (namespace ito) 3](#_Toc337826481)

[Test 1 3](#_Toc337826482)

[Test 2 3](#_Toc337826483)

[Test 3 3](#_Toc337826484)

[Test 4 4](#_Toc337826485)

[Test 5 4](#_Toc337826486)

[Test 6 4](#_Toc337826487)

[DataObject 5](#_Toc337826488)

[Test 1 5](#_Toc337826489)

[Test 2 5](#_Toc337826490)

# How to read this document

This section is mainly divided into tests for different classes. In every class there will be a set of different main tests (Test 1, Test 2…). Every test may have an initialization, where an object can be initialized, followed by an execution section, where lots of different tests are executed. These tests work on the object(s) created in the init-section.

# Class Range

## Test 1

**Init**

-  
  
**Execution**

* Range() 🡪 member *start* and *end* must be 0
* Range(s,e) [s<=e, Int] 🡪 *start* and *end* must correspond to *s,e* respectively
* Range(s,e) [s>e, Int] 🡪 *start* and *end* must correspond to *e,s* respectively
* bool empty() 🡪 must return *true* if *start==end*, else *false*
* static all() 🡪 Range-object must be created with *start==INT\_MIN* and *end==INT\_MAX*
* copy constructor 🡪 check that members *start* and *end* are copied
* operator = 🡪 check that members *start* and *end* are copied

# Class DataObjectTagType

## Test 1

**Init**

Create empty DataObjectTagType (“*obj = DataObjectTagType()*”)

**Execution**

* *getType()* must return *tTagType::typeInvalid*
* *isValid()* must return *false*
* *getVal\_ToDouble()* must return *std::numeric\_limits<double>::signaling\_NaN()*
* *getVal\_ToString()* must return empty *std::string()*
* Call copy constructor for this object and check the tests above also for this copied object
* Call operator= for this object and check the tests above also for this assigned object

## Test 2

**Init**

* Create DataObjectTagType with any finite double value (obj1)
* Create obj2 with *std::numeric\_limits<double>::quiet\_NaN()*
* Create obj3 with *std::numeric\_limits<double>::signaling\_NaN()*
* Create obj4 with *std::numeric\_limits<double>::infinity()*

**Execution**

* *getType()* must return *tTagType::typeDouble* for *obj1..obj4*
* *isValid()* must return *true* for *obj1..obj4*
* *getVal\_ToDouble()* must return the same value for *obj1..obj4*
* *getVal\_ToString()*must return the string-representation of this double value (obtained by *std::ostringstream s; s << doubleValue;*) for *obj1*, “NaN” for *obj2* and *obj3* and “Inf” for *obj4*.
* Call copy constructor for this object and check the tests above also for this copied object
* Call operator= for this object and check the tests above also for this assigned object

## Test 3

**Init**

Create DataObjectTagType with any string

**Execution**

* *getType()* must return *tTagType::typeString*
* *isValid()* must return *true*
* *getVal\_ToDouble()* must return *std::numeric\_limits<double>::signaling\_NaN()*
* *getVal\_ToString()*must return the same string
* Call copy constructor for this object and check the tests above also for this copied object
* Call operator= for this object and check the tests above also for this assigned object

# General methods in dataobj.h (namespace ito)

## Test 1

This test should test the method ***isZeroValue<\_Tp>(…)***

**Init**

-

**Execution**

* *isZeroValue()* for types ito::(u)int8, ito::(u)int16, ito::int32. Value 0 must return *true*, value != 0 (try 2-4 different values) must return *false*
* *isZeroValue()* for types ito::float32, ito::float64 must return *true*, if value is in range (‑*std::numeric\_limits<type>::epsilon, std::numeric\_limits<type>::epsilon*), else *false*. Boundaries of range result in *false*. Check it for different values: 0, in between range, range boundaries, slightly outside of range and far away.
* *isZeroValue()* for types ito::complex64 and ito::complex128 only result *true*, if both real and imaginary part is equal to zero. Check if for some combinations of different real and imaginary parts.

## Test 2

This test should test the method ***convertCmplxTypeToRealType(…)***

**Init**

-

**Execution**

Call the method for types…

* ito::t(U)Int[8/16/32] must return the same type constant
* ito::tFloat32 and ito::tFloat64 must also return the same type constant
* ito::tComplex64 must return ito::tFloat32
* ito::tComplex128 must return ito::Float64

## Test 3

This test should test the method ***numberConversion<type>(…)***

**Init**

From every type in ito::tDataType create the following variables:

* -5, 0, 5 for fixed-point types
* -5.0, -4.9, -4.1, 0, 4.1, 4.9, 5.0 for floating-point values
* -5.0+5.0i, -5.0+0i, 0.0-5.0i, 0.0+5.0i,5.0-5.0i for complex values

**Execution**

Call the method for types…

* Convert every of these values to every available data type:
  + The fixed point values must be the same if converted to other fixed-point values, they must be the same than the floating-point values, considering the epsilon-value and finally if converted to complexed values, the imaginary part must be 0, while the real part is the same than the conversion to the corresponding floating type value
  + The floating-point values: Conversion to complex: real part must be the same while imaginary part must be zero, conversion to floating point: must be the same, conversion to fixed-point: value must be same considering that no rounding is executed by simple cutting of the decimal values.
  + Complex-Values can only be converted to other complex-values, else an exception is raised and must be checked (that it is raised)

## Test 4

This test should test the method ***saturate\_cast<type>(…), namespace cv***

**Init**

From every type in ito::tDataType create the following variables:

* -5, 0, 5 for fixed-point types
* -5.0, -4.9, -4.1, 0, 4.1, 4.9, 5.0 for floating-point values
* -5.0+5.0i, -5.0+0i, 0.0-5.0i, 0.0+5.0i,5.0-5.0i for complex values
* Additionally create the minimum and maximum allowed value for this type (using std::numeric\_limits) and values (minimum+10 and maximum-10)
* For floating point values (float32 and float64) create infinity(), signaling\_NaN() and quiet\_NaN() (**special values**)

**Execution**

* Convert every given value to any type (ito::(u)int8, ito::(u)int16, ito::int32, ito::float[32/64], ito::complex[64/128].
* The following rules need to be checked:
  + Decimal is cut and not rounded from floating to fixed-point
  + If value exceeds range of destination value, the corresponding boundary of destination value is returned
  + Special values remain the same special values when converting between float32 and float64 or vice-versa; when converting from floating to fixed-point, the maximum boundary of the fixed-point value must be returned
  + Cast from complex to non-complex throws an error
  + Cast from non-complex to complex results in a complex, where real corresponds to non-complex value and imaginary part is equal to zero.

## Test 5

This test should test the method ***getDataType(…)***

**Init**

-

**Execution**

* + Test whether getDataType, called with a variable of any pointer type, corresponding of values of ito::tDataType, returns the corresponding element of ito::tDataType (e.g. getDataType( (const ito::uint8\*)NULL) should return ito::tUInt8.

## Test 6

This test should test the method ***getDataType2<\_Tp>()***

**Init**

-

**Execution**

* + Test whether getDataType2, called with a template parameter of any pointer type, corresponding of values of ito::tDataType, returns the corresponding element of ito::tDataType (e.g. getDataType<uint8\*>() should return ito::tUInt8.

# DataObject

## Test 1

This test checks the methods **seekMat** and **calcNumMats**

**Init**

Create:

* Empty data object *dObj1*
* Two-dimensional data object *dObj2* of size 10x10 (type, values arbitrary)
* Three-dimensional data object *dObj3* of size 5x10x10 (type, values arbitrary)
* *dObj4* = *dObj3(2:4,:,:)*
* five-dimensional data object *dObj5* of size 3x4x2x10x10 (type, values arbitrary)
* *dObj6 = dObj5(1:3,2:3,:,:,:)*

**Execute**

* dObj1.seekMat(i) must be 0 (invalid) for i=0..4
* dObj2.seekMat(i) must give [0,0,0,0,0] for i=0..4
* dObj3.seekMat(i) must give [0,1,2,3,4,0] for i=0..5
* dObj4.seekMat(i) must give [2,3,0,0,0] for i=0..5
* dObj5.seekMat(i) must give [0,1,2,…,23,0,0,…] for i=0..30
* dObj6.seekMat(i) must give [12,13,20,21,0,0,…0] for i=0..30
* dObj1.calcNumMats() must return 0
* dObj2.calcNumMats() must return 1
* dObj3.calcNumMats() must return 5
* dObj4.calcNumMats() must return 2
* dObj5.calcNumMats() must return 24
* dObj6.calcNumMats() must return 4

## Test 2

This test checks the methods **getDims** and **getType**

**Init**

Create for all possible types (templating):

* Empty data object *dObj1*
* One-dimensional data object dObj2 (use the n-dim constructor for that, size arbitrary, values arbitrary)
* Two-dimensional data object dObj3
* Three-dimensional data object dObj4
* Five-dimensional data object dObj5

**Execute**

* dObj1.getDims() must be 0, getType() must correspond to your type
* dObj2.getDims() must be **2** (dim 1 does not exist) , getType() must correspond to your type
* dObj3.getDims() must be 2, getType() must correspond to your type
* dObj4.getDims() must be 3, getType() must correspond to your type
* dObj5.getDims() must be 5, getType() must correspond to your type

## Test 4

This test checks the default values of the tags.

**Init**

Create for all possible types (templating):

* Empty data object *dObj1,* dim 0
* Two-dimensional data object dObj2, dim 2
* Three-dimensional data object dObj3, dim 3
* Five-dimensional data object dObj4, dim 5
* DataObject dObj5(dObj2) – copy constructor, dim 2
* DataObject dObj6 = dObj2 – assignment operator (=), dim 2
* dObj7: One-dimensional data object dObj1, dim 1

**Execute**

* Method *getValueOffset*()must return **0.0** for all dataObjects
* Method *getValueScale()* must return **1.0** for all dataObjects
* Method *getValueUnit()* must return std::string() for all dataObject
* Method *getValueDescription()* must return std::string() for all dataObject
* dObj1.getAxisOffset(i) must throw an exception for i=(-1,0,1)
* dObj1.getAxisScale(i) must throw an exception for i=(-1,0,1)
* dObj2-6.getAxisOffset(i) must return (exc,0.0,..,0.0,exc) for i=(-1,0,1,..,dims, dims+1)
* dObj2-6.getAxisScale(i) must return (exc,1.0,..,1.0,exc) for i=(-1,0,1,..,dims, dims+1)
* dObj7.getValueOffset() must return 0.0
* dObj7.getValueScale() must return 1.0
* dObj7.getValueUnit() must return std::string()
* dObj7.getValueDescription() must return std::string()
* dObj7.getAxisOffset(i) must return (exc,0.0,0.0,exc) for i=(-1,0,1,2)
* dObj7.getAxisScale(i) must return (exc,1.0,1.0,exc) for i=(-1,0,1,2)
* dObj1-7.getXYRotationalMatrix(…) must return 1,0,0,0,1,0,0,0,1 (3x3-eye-matrix-values)

**Warning**

The exception behavior has been changed and must be adapted in the existing test-cases.

## Test 5

This tests checks the method **copyTo**

**Task**

Execute the copyTo-command with regionOnly-parameter set to **true** and **false**. Use the following source dataObjects:

* Empty DataObject
* 2, 3 and 5 dimensional dataObject with full-ROI
* 2, 3 and 5 dimensional dataObject with ROI smaller than the maximum size

Use the following destination dataObjects:

* Empty DataObject
* 2, 3 and 5 dimensional dataObject with full-ROI
* 2, 3 and 5 dimensional dataObject with ROI smaller than the maximum size

Verify that:

* regionOnly=true: The destination object has the same size and number of dimensions than the ROI of the source. Values in ROI of source and destination are equal.
* regionOnly = false: The destination object has the same original size and size than the source. ROI must have the same position and size, hence try to locate roi in source and destination and compare it. Adjust ROI of source and destination, such that the original matrix size is obtained. All values must be equal.

## Test 6

This tests checks the method **deepCopyPartial**

**Task**

Execute the **deepCopyPartial**-command. Use the following source dataObjects:

* Empty DataObject
* 2, 3 and 5 dimensional dataObject with full-ROI
* 2, 3 and 5 dimensional dataObject with ROI smaller than the maximum size

Use the following destination dataObjects:

* Empty DataObject
* 2, 3 and 5 dimensional dataObject with full-ROI
* 2, 3 and 5 dimensional dataObject with ROI smaller than the maximum size, but the ROI with the same size and same position than the equivalent source-dataObject.
* 2, 3 and 5 dimensional dataObject with ROI smaller than the maximum size, but the ROI with the same size but different position than the equivalent source-dataObject.
* 2, 3 and 5 dimensional dataObject with ROI smaller than the maximum size and different than the ROI of the corresponding source-dataObject.

Verify that:

* If size (of ROI) of source and destination are unequal exception must be raised
* If type of source and destination are unequal exception must be raised
* Values of ROI of source must be equal to values in ROI of destination after copying.
* Values outside of ROI of destination must remain equal (access these values by adjusting-ROI of destination)

## Test 7

This tests checks the method **adjustROI**

**Init**

Implement these tests for a 0,2,3 and 5-dimensional dataObject. They all have different, but characteristic behaviours.

**Task**

For each dataObject, try to adjust the ROI with the following properties:

* Call adjustROI with the four-parameter implementation and the general implementation.
* Call adjustROI with the wrong number of arguments -> exception must be raised
* Call adjustROI with values, such that the resulting ROI lies 1. Within the valid matrix-region, 2. Partially outside the valid matrix-region, 3. Completely outside the valid matrix-region, 4. Such that the resulting ROI has a “negative” area. For cases 2-4, an exception must be raised, for section 1, check that the values within the ROI are equal to the expected values

This method has recently been changed. In the past, it had some unwanted behavior, which has not been rejected by the recent test-case. In the past, a ROI larger than the possible ROI has been trimmed to the biggest possible size, now an exception is raised. Please intensively check for this exception.

## Test 8

This tests checks the method **at(ito::Range \*ranges)**

This method is similar to adjustROI and internally calls adjustROI. Please check it with similar matrices than adjustROI and check for exceptions, which must be raised, if the indicates ranges exceed the possible size of the matrix. Additionally, check, that if ranges contains a Range::all() value, the corresponding dimension is not changed in size.

This method has recently been changed. In the past, it had some unwanted behavior, which has not been rejected by the recent test-case.

## Test 9

This tests checks the method **locateROI**

Use the tests 7 and 8 in order to also check locateROI. Now, locateROI knows two implementations, where the second (one argument only) not has been tested by the old unittest-framework. The result of this one-argument-implementation should return exactly the same result than the input to the adjustROI method, if the matrix had its original, biggest possible size, before.