Introduction to ImarisXT and IceImarisConnector

Extending Imaris with custom MATLAB functions

Version 1.1.0

Aaron Ponti

Contents

1 Introduction

Even the best software packages for data analysis cannot possibly offer all of the functionalities required for solving the problem at hand. Indeed, problem-specific customizations are necessary in all but the most trivial cases. Bitplane Imaris[1] allows to extend its palette of built-in processing functions through ImarisXT[2], a two-way interface to classic programming languages (such as Java, C++, C# and Visual Basic), MATLAB[3] and other image processing tools like ImageJ[4] and Fiji[5].

Historically, ImarisXT used Microsoft's Component Object Model (COM)[6] binary interface to allow interprocess communication with external tools; after Imaris was ported to Mac OS X, COM (which only runs on Windows) was replaced with the Internet Communication Engine (Ice)[7], an efficient, cross-platform, distributed computing framework. ImarisXT exposes Ice through a Java wrapper API. While Ice has made it possible for Imaris users on Mac OS X to use ImarisXT on their favorite platform, the COM-based interface was easier to use and performed a lot of work behind the scenes that is now responsibility of the user. We will discuss several of these issues during the course.

By far the most common use of ImarisXT is in combination with MATLAB, and indeed ImarisXT is bundled with a long list of already implemented MATLAB scripts (so called *XTensions*) that perform all sort of tasks. Recently, Imaris added easy integration of ImageJ / Fiji and their plug-ins and, as of Imaris 7.6, support for python[8].

While we will take a quick look on how to use existing Fiji plug-ins and how to add new ones, the bulk of this course will teach how to write Imaris *XTensions* in MATLAB and will present a free and open-source library that facilitates the usage of ImarisXT: `IceImarisConnector`[9].

1.1 Bundled MATLAB *XTensions*

The MATLAB *XTensions* bundled with ImarisXT[10] can be started from the *Image Processing* menu (see figure 1.1). They appear in the menu with a small MATLAB icon next to the name.

[1] http://www.bitplane.com
[2] http://www.bitplane.com/go/products/imarisxt
[3] http://www.mathworks.com
[4] http://rsbweb.nih.gov/ij/
[5] http://fiji.sc
[6] http://en.wikipedia.org/wiki/Component_Object_Model
[7] http://www.zeroc.com
[8] http://www.python.org
[9] http://www.scs2.net/next/index.php?id=110, which incidentally is also the home of the python alter ego of IceImarisConnector: pIceImarisConnector (a.k.a. IceImarisConnector for python)
[10] Please notice that in recent Imaris versions, the bundled XTensions are not active by default. A configuration step is required, as explained in section 1.4.

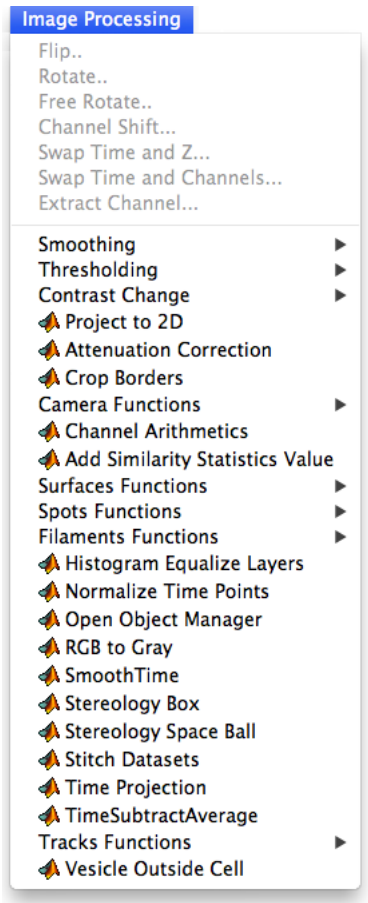


Figure 1.1: The bundled MATLAB *XTensions* in the “Image processing” menu.

When launching an *XTension*, Imaris will start MATLAB in the background which in turn will run it. Even though the function will run behind the scenes, interactivity is possible: for example, the *XTension* might open dialogs to ask for input parameters. Finally, the *XTension* can send any results of the computation back to Imaris, display them on plots and figures, save them to disk, or open them in Excel.

The Imaris reference manual contains a short help for each of the bundled *XTensions* (see figures 1.2 and 1.3). As we will see in section 4, Imaris also provides good documentation on how to write our own *XTensions* (see figure 4.2). Another great reference is provided by the source code of the bundled *XTensions*.

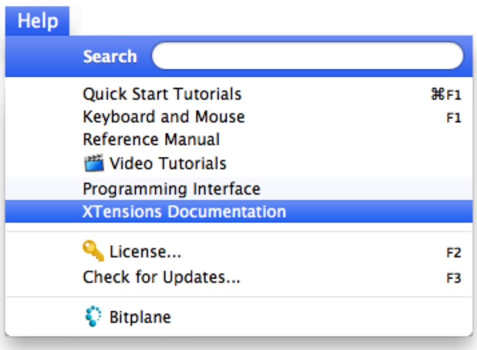


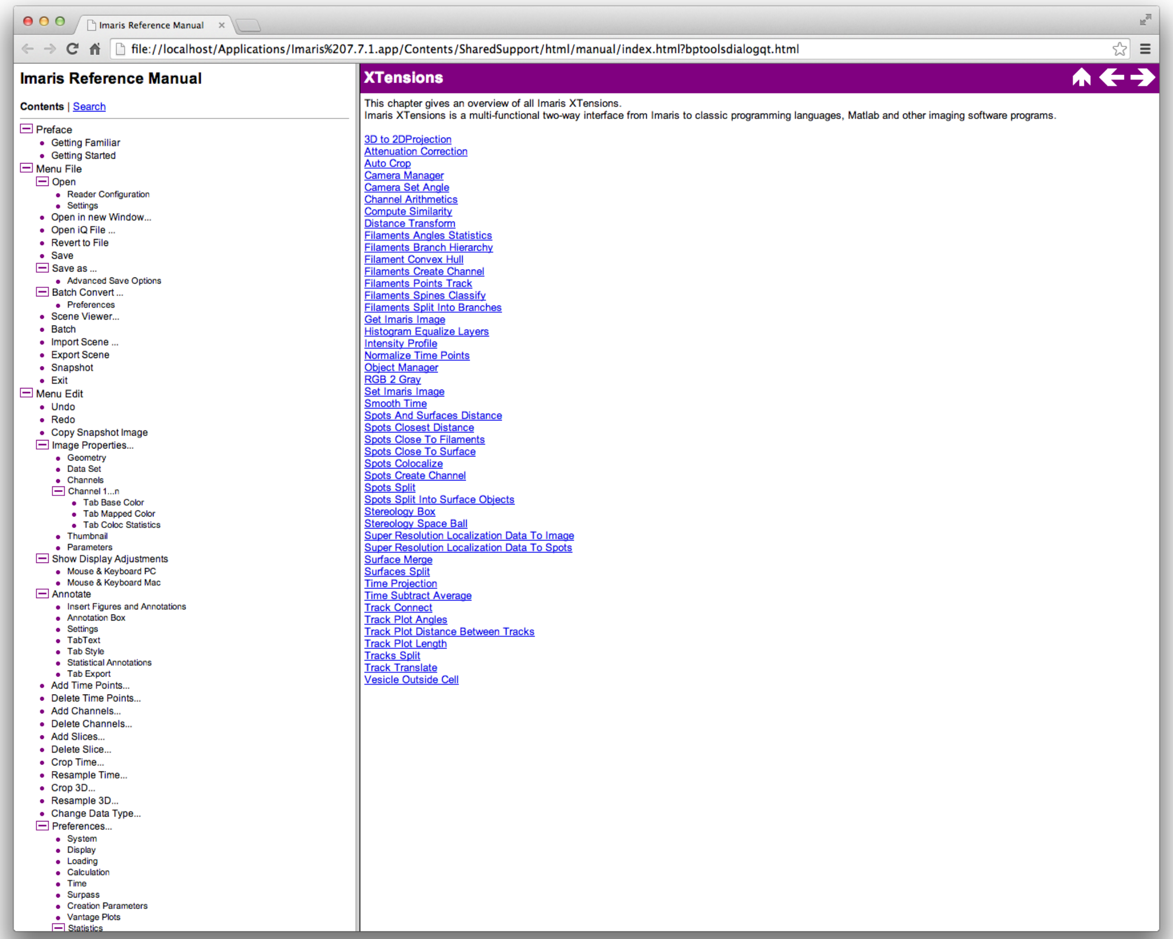
Figure 1.2: Link to the documentation about the bundled MATLAB *XTensions*.

1.2 Fiji methods and plug-ins

As a more recent alternative to interfacing to MATLAB, ImarisXT now also allows inter-process communication with ImageJ or Fiji. Fiji is a well established image processing tool with a good number of integrated functionalities and an impressive number of plug-ins provided by the community, and ImarisXT makes it easy to integrate this palette of additional functionalities into the Imaris working environment.

The number of Fiji plug-ins that come bundled with Imaris is not as impressive as the number of MATLAB *XTensions*, but a few examples exist and can be accessed from the Fiji menu. In addition to the examples in the *Process* and *Plugins* submenu, *Image From Fiji* and *Image To Fiji* allow for a very easy transfer of datasets from Fiji to Imaris and viceversa (see figure 1.4).

The first time a Fiji plug-in is launched from Imaris, the ImarisBridge must be installed (see figure 1.5): this enables the two-way communication between the two environments.



Imaris Reference Manual

file://localhost/Applications/Imaris%207.7.1.app/Contents/SharedSupport/html/manual/index.html?bptoolsdialogqt.html

Imaris Reference Manual

Contents | Search

- Preface
  - Getting Familiar
  - Getting Started
- Menu File
  - Open
    - Reader Configuration
    - Settings
  - Open in new Window...
  - Open iQ File ...
  - Revert to File
  - Save
  - Save as ...
    - Advanced Save Options
  - Batch Convert ...
    - Preferences
  - Scene Viewer...
  - Batch
  - Import Scene ...
  - Export Scene
  - Snapshot
  - Exit
- Menu Edit
  - Undo
  - Redo
  - Copy Snapshot Image
  - Image Properties...
    - Geometry
    - Data Set
    - Channels
    - Channel 1...n
      - Tab Base Color
      - Tab Mapped Color
      - Tab Coloc Statistics
    - Thumbnail
    - Parameters
  - Show Display Adjustments
    - Mouse & Keyboard PC
    - Mouse & Keyboard Mac
  - Annotate
    - Insert Figures and Annotations
    - Annotation Box
    - Settings
    - TabText
    - Tab Style
    - Statistical Annotations
    - Tab Export
  - Add Time Points...
  - Delete Time Points...
  - Add Channels...
  - Delete Channels...
  - Add Slices...
  - Delete Slice...
  - Crop Time...
  - Resample Time...
  - Crop 3D...
  - Resample 3D...
  - Change Data Type...
  - Preferences...
    - System
    - Display
    - Loading
    - Calculation
    - Time
    - Surpass
    - Creation Parameters
    - Vantage Plots
    - Statistics

XTensions

This chapter gives an overview of all Imaris XTensions.
Imaris XTensions is a multi-functional two-way interface from Imaris to classic programming languages, Matlab and other imaging software programs.

3D to 2DProjection
Attenuation Correction
Auto Crop
Camera Manager
Camera Set Angle
Channel Arithmetics
Compute Similarity
Distance Transform
Filaments Angles Statistics
Filaments Branch Hierarchy
Filament Convex Hull
Filaments Create Channel
Filaments Points Track
Filaments Spines Classify
Filaments Split Into Branches
Get Imaris Image
Histogram Equalize Layers
Intensity Profile
Normalize Time Points
Object Manager
RGB 2 Gray
Set Imaris Image
Smooth Time
Spots And Surfaces Distance
Spots Closest Distance
Spots Close To Filaments
Spots Close To Surface
Spots Colocalize
Spots Create Channel
Spots Split
Spots Split Into Surface Objects
Stereology Box
Stereology Space Ball
Super Resolution Localization Data To Image
Super Resolution Localization Data To Spots
Surface Merge
Surfaces Split
Time Projection
Time Subtract Average
Track Connect
Track Plot Angles
Track Plot Distance Between Tracks
Track Plot Length
Tracks Split
Track Translate
Vesicle Outside Cell

Figure 1.3: The bundled MATLAB *XTensions* explained.

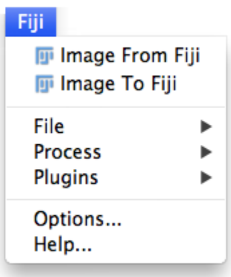


Figure 1.4: The “Fiji” menu.

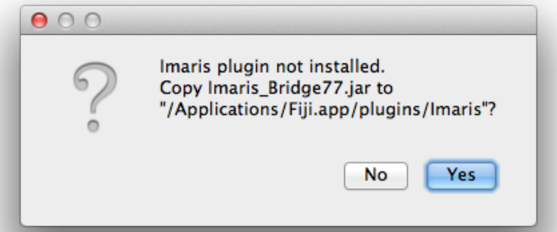

Imaris plugin not installed.
Copy Imaris_Bridge77.jar to
"/Applications/Fiji.app/plugins/Imaris"?

Figure 1.5: The communication between Imaris and Fiji requires the installation of a special Imaris Bridge.

1.3 Bundled python *XTensions*

As of Imaris 7.6, ImarisXT also supports writing *XTensions* in the python programming language[11] in its 2.7.x version. In Imaris versions 7.6 and 7.7, only three simple example *XTensions* are bundled as templates. Writing python *XTensions* will not be treated in this course; an example *XTension* in python is shown in section 3.

1.4 Setup

1.4.1 Application paths

To ensure that Imaris can start MATLAB when the user launches an *XTension*, a setup step is required. From the *Imaris* (Mac OS X) or *Edit* menu (Windows), choose *Preferences* and change to the *Custom Tools* pane (see figure 1.6).

Make sure to set the path the MATLAB executable. On Windows[12] it will be something like:

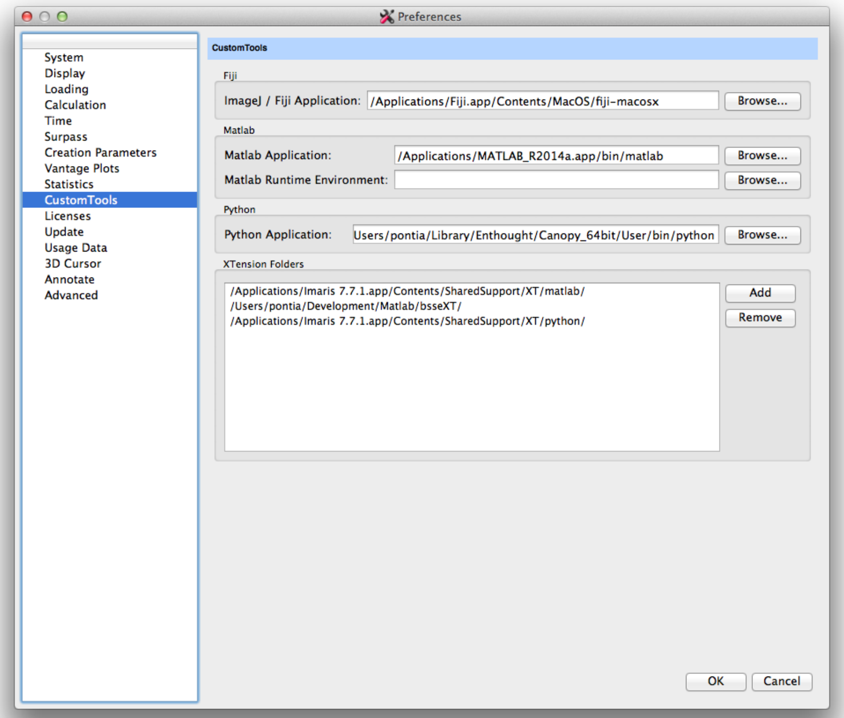
```
C:\Program Files\MATLAB\R201Xa_x64\bin\win64\MATLAB.exe
```

while on Mac OS X it will be something like:

```
/Applications/MATLAB_R201Xa.app/bin/matlab
```

If you do not own a license for MATLAB, the MATLAB COMPILER RUNTIME (referred to as MATLAB Runtime Environment in the Custom Tools dialog) can be used instead. The MCR is free and can be downloaded from the MathWorks website[13]. Please mind that while this will allow the bundled *XTensions* to run, you will not be able to write and run your own custom *XTensions*!

[11]http://www.python.org.
[12]No longer needed in recent Imaris versions.
[13]http://www.mathworks.com/products/compiler/mcr/index.html



CustomTools

Fiji

ImageJ / Fiji Application: /Applications/Fiji.app/Contents/MacOS/fiji-macosx

Matlab

Matlab Application: /Applications/MATLAB_R2014a.app/bin/matlab

Matlab Runtime Environment:

Python

Python Application: Users/pontia/Library/Enthought/Canopy_64bit/User/bin/python

XTension Folders

/Applications/Imaris 7.7.1.app/Contents/SharedSupport/XT/matlab/
/Users/pontia/Development/Matlab/bsseXT/
/Applications/Imaris 7.7.1.app/Contents/SharedSupport/XT/python/

Figure 1.6: The Custom Tools preferences pane.

As for MATLAB, also the ImageJ / Fiji Application path must be set (see Figure 1.6). Since Fiji does not come with an installer, the path will depend on where you extracted the archive. Analogously, to run python *XTensions*, python 2.7 must be installed and the path to the python executable must be set.

1.4.2 *XTension* folders

The paths to all *XTensions* folders[14] was hard-coded in Imaris up to version 7.5 to

```
C:\Program Files\Bitplane\Imaris 7.x.y\XT\{tool}
```

on Windows and

```
/Applications/Imaris 7.x.y.app/Contents/SharedSupport/XT/{tool}
```

on Mac OS X, with *{tool}* mapping to a set of subfolder names as follows:

rtmatlab	for the compiled MATLAB *XTensions* to be used with the MATLAB COMPILER RUNTIME;
matlab	for the standard MATLAB *XTensions* to be used with MATLAB;
fiji	for the Fiji/ImageJ *XTensions*.

In more recent versions of Imaris, those subfolders still exist, with the notable addition of:

python	for the python *XTensions,*

however, Imaris ignores them until they are explicitly configured in the *XTensions Folders* field of the Custom Tools preference pane (see figure 1.6 at the bottom). Additional, custom *XTension* folders must be added there as well.

[14] *For both* bundled and custom *XTensions*.

2 Interfacing with Fiji

Adding a plug-in to the Imaris Fji menu is done via short text files that contain both the information on where they are to be installed in the Fiji menu and the actual code. As an example, navigate to:

```
/Applications/Imaris 7.x.y.app/Contents/SharedSupport/XT/fiji
```

on Mac OS X or

```
C:\Program Files\Bitplane\Imaris 7.x.y\XT\fiji
```

on Windows, and open one of the text files you find there. Here, we will take a look at the *EEE_Process_Find Edges.txt* file[1].

The first part of the file is a header that Imaris uses to add the entry to the Fiji menu:

```
// <CustomTools>
//  <Menu name="Fiji">
//   <Submenu name="Process">
//    <Item name="Find Edges" icon="Fiji">
//     <Command>ImageJ::EEE_Process_Find Edges</Command>
//    </Item>
//   </Submenu>
//  </Menu>
// </CustomTools>
```

The header informs Imaris that a custom tool with name "Find Edges" will be added to the *Fiji* menu, in the submenu *Process* and will run the (ImageJ macro) code in the file `EEE_Process_Find Edges.txt` file. The actual code:

```
call("Imaris_Bridge.In", getArgument());
run("Find Edges");
call("Imaris_Bridge.Out", getArgument());
```

will run the *In()* method from the *Imaris_Bridge.in* Java jar package passing the reference to the current active Imaris dataset to be copied to Fiji, will then run the "Find Edges" method in Fiji, and copy back the result from the active Fiji window to Imaris. The code between to two calls to the Imaris_Bridge is simple ImageJ macro language[2].

As another example, we will write our own Fiji *XTension* named:

```
HHH_ImarisXT_Course_AboutIJ.txt,
```

[1]The purpose of the uppercase letters at the beginning of the names (i.e. *EEE_*) is just for sorting.
[2]http://rsbweb.nih.gov/ij/developer/macro/macros.html

that simply opens the “About ImageJ” image in Fiji and copies it to Imaris.

```
// <CustomTools>
//  <Menu name="Fiji">
//   <Submenu name="ImarisXT Course">
//    <Item name="About ImageJ" icon="Fiji">
//     <Command>ImageJ::HHH_ImarisXT_Course_AboutIJ</Command>
//    </Item>
//   </Submenu>
//  </Menu>
// </CustomTools>

run("About ImageJ...");
call("Imaris_Bridge.Out", getArgument());
close();
```

In our header, we add it to an “ImarisXT Course” submenu straight under the Fiji menu.

3 An example python *XTension*

Writing python *XTensions* is not treated in this course: we will just look at one of the bundled examples. After reading chapter 4, you will find several parallels between python and MATLAB *XTensions*.

```
#
#
# PythonXT Simple Spots Example for Imaris
#
#
# Copyright Bitplane AG
#
#
# <CustomTools>
#  <Menu>
#   <Item name="Simple Spots Example" icon="Python" tooltip="Simple XTension... Dataset">
#    <Command>PythonXT::XTSimpleSpotsExample(%i)</Command>
#   </Item>
#  </Menu>
# </CustomTools>

import time
import ImarisLib

def XTSimpleSpotsExample(aImarisId):

  # Create an ImarisLib object
  vImarisLib = ImarisLib.ImarisLib()

  # Get an imaris object with id aImarisId
  vImaris = vImarisLib.GetApplication(aImarisId)

  # Check if the object is valid
  if vImaris is None:
    print 'Could not connect to Imaris!'

    # Sleep 2 seconds to give the user a chance to see the printed message
    time.sleep(2)
    return

  # Get the currently loaded dataset
  vImage = vImaris.GetDataSet()

  # This xtension requires a loaded dataset
  if vImage is None:
    print 'An image has to be loaded into Imaris first'
```

```
      time.sleep(2)
      return

    # Get the extents of the image
    vExtentMinX = vImage.GetExtendMinX();
    vExtentMinY = vImage.GetExtendMinY();
    vExtentMinZ = vImage.GetExtendMinZ();
    vExtentMaxX = vImage.GetExtendMaxX();
    vExtentMaxY = vImage.GetExtendMaxY();
    vExtentMaxZ = vImage.GetExtendMaxZ();
    vImageSizeX = vExtentMaxX - vExtentMinX;
    vImageSizeY = vExtentMaxY - vExtentMinY;
    vImageSizeZ = vExtentMaxZ - vExtentMinZ;
    vMinRadius = min(vImageSizeX, vImageSizeY, vImageSizeZ) / 2;

    # Calculate the center of the image
    vCenterX = (vExtentMinX + vExtentMaxX) / 2;
    vCenterY = (vExtentMinY + vExtentMaxY) / 2;
    vCenterZ = (vExtentMinZ + vExtentMaxZ) / 2;

    # Create a spots component
    vSpotsComponent = vImaris.GetFactory().CreateSpots();

    # Add one spot in the center
    # The positions array is 2 dimensional
    vSpotsComponent.Set([[vCenterX, vCenterY, vCenterZ]], [0], [vMinRadius]);

    # Give the component a nice name
    vSpotsComponent.SetName('Image Center Spot');

    # Color the one spot red
    vSpotsComponent.SetColorRGBA(255);

    # Add the spots component at the end of the surpass tree
    vImaris.GetSurpassScene().AddChild(vSpotsComponent, -1);
```

Although writing Imaris *XTensions* in python is not treated in this course, it is worth pointing out that there exists a companion to the IceImarisConnector that facilitates writing Imaris XTensions with python: *pIceImarisConnector*[1] (IceImarisConnector for python).

[1]http://www.scs2.net/next/index.php?id=110

4 Writing MATLAB *XTensions*

In this section, we will write two MATLAB *XTensions* using exclusively the standard API. In the next section we will rewrite the same *XTensions* taking advantage of the additional functionalities provided by IceImarisConnector.

4.1 Getting the course code

In preparation to it, go ahead and create a folder on your hard disk where you will store the four MATLAB *XTensions* and add it to the *XTension* folders as discussed in section 1.4.2 and shown on figure 1.6. You can download the already written *XTensions*[1] and an Imaris file[2] that we will need for testing them from http://www.scs2.net/next/index.php?id=20.

4.2 Documentation

The documentation of ImarisXT consists of its API (see figure 4.1). All classes of the XT interface are thoroughly documented. In the example of figure 4.2, all methods of the `ISpots` class are displayed. An `ISpots` object *represents* the spots created in the Surpass Scene and allows to manipulate them programmatically as we will see in later sections. Each method signature in Figure 4.2 is a link to more detailed information and some example code.

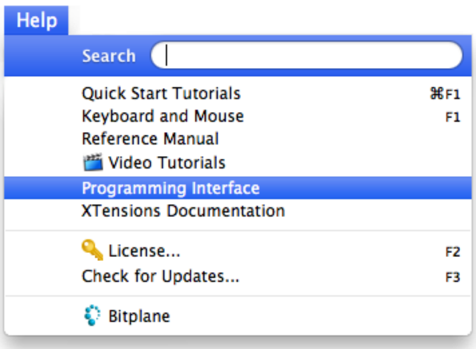
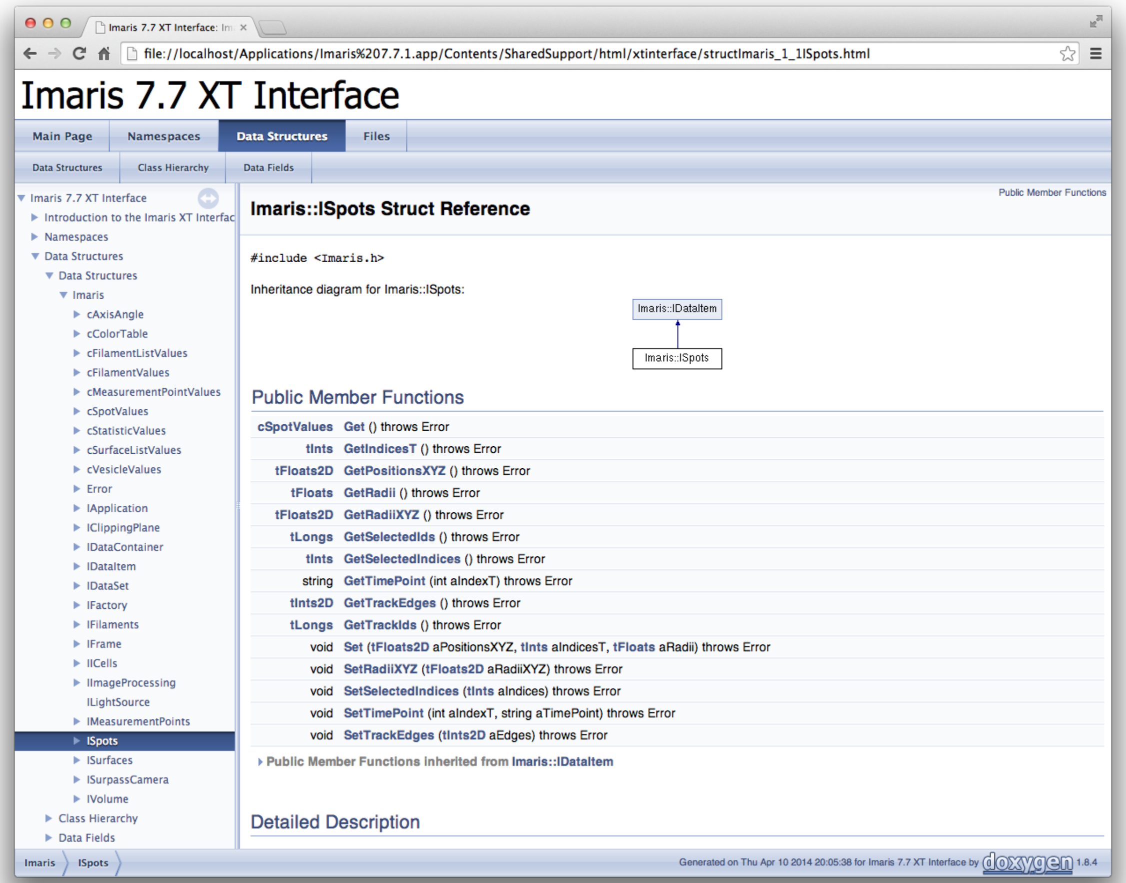


Figure 4.1: The link to the ImarisXT API documentation in the Help menu.

[1]Direct link: http://www.scs2.net/next/files/courses/iic/XTensions.zip
[2]Direct link: http://www.scs2.net/next/files/courses/iic/ImarisXTCourse.ims.zip



Imaris 7.7 XT Interface

Imaris::ISpots Struct Reference

```
#include <Imaris.h>
```

Inheritance diagram for Imaris::ISpots:

Public Member Functions

cSpotValues	Get () throws Error
tInts	GetIndicesT () throws Error
tFloats2D	GetPositionsXYZ () throws Error
tFloats	GetRadii () throws Error
tFloats2D	GetRadiiXYZ () throws Error
tLongs	GetSelectedIds () throws Error
tInts	GetSelectedIndices () throws Error
string	GetTimePoint (int aIndexT) throws Error
tInts2D	GetTrackEdges () throws Error
tLongs	GetTrackIds () throws Error
void	Set (tFloats2D aPositionsXYZ, tInts aIndicesT, tFloats aRadii) throws Error
void	SetRadiiXYZ (tFloats2D aRadiiXYZ) throws Error
void	SetSelectedIndices (tInts aIndices) throws Error
void	SetTimePoint (int aIndexT, string aTimePoint) throws Error
void	SetTrackEdges (tInts2D aEdges) throws Error

Public Member Functions inherited from Imaris::IDataItem

Detailed Description

Figure 4.2: The ImarisXT API documentation.

4.3 ImarisXT pitfalls

The document `Matlab Migration Guide To Imaris73.pdf` found in the `XT/matlab` folder contains a detailed treatment of all differences from the old ImarisXT-based COM interface to the new Ice-based one, and lists rather succinctly several limitations of the usage of Java for the communication between Imaris and MATLAB. The document does not go into any significant details, therefore we will discuss some examples here that require particular care.

4.3.1 MATLAB closes when the *XTension* finishes

The COM-based ImarisXT would start MATLAB and maintain a live connection until the user would close either one of the software. This means that once MATLAB had started, all subsequent *XTensions* could be run immediately. The Ice-based ImarisXT instead closes MATLAB as soon as the *XTension* has finished executing. Possibly the most important drawbacks of forcibly closing MATLAB at the end of each run is that debugging *XTensions* becomes quite complicated (albeit not impossible); moreover, in case of lengthy initialization scripts, MATLAB might take a while to start, thus introducing a delay between repeating runs. Trying to trick MATLAB into staying open by displaying a modal dialog or a figure[3] will not work, since MATLAB will stay open but will not be usable., since the open figure blocks.

4.3.2 Signed vs. unsigned integers

All integer types in Java are signed. An 8-bit integer in MATLAB can be signed (and cover the range -128:127) or unsigned (and cover the range 0:255); similarly, a 16-bit integer in MATLAB is either signed (-32768:32767) or unsigned (0:65535). Images are usually stored as 8- or 16-bit unsigned integers (or in some rare cases a 32-bit float, but this is irrelevant for our discussion here), and this is how both MATLAB and Imaris represents them internally. The tricky part is the *transfer* of image data from Imaris to MATLAB and vice-versa, which happens through Java[4].

4.3.2.1 Getting image data from Imaris to MATLAB

While COM offered one function that would take care of transferring the data with the correct datatype (`IDataSet::GetDataVolume()`), with Ice all the work is left to the user. We will discuss the details of the code later, here we will just show the steps required to get the 3D stack for the first channel and first time point:

```
% Get the dataset class
switch char(iDataset.GetType())
    case 'eTypeUInt8',   datatype = 'uint8';
    case 'eTypeUInt16',  datatype = 'uint16';
    case 'eTypeFloat',   datatype = 'single';
    otherwise,           error('Bad value for iDataSet.GetType()');
end
```

[3]As a general recommendation, if your *XTension* generates plots, better save them to disk from your code.
[4]This is actually a problem wherever integers are involved, like for instance when getting/setting RGBA colors.

```
% Allocate memory
stack = zeros([iDataset.GetSizeX(), iDataset.GetSizeY(), ...
    iDataset.GetSizeZ()], datatype);

% Get the stack
switch char(iDataset.GetType())
    case 'eTypeUInt8',
        arr = iDataset.GetDataVolumeAs1DArrayBytes(0, 0);
        stack(:) = typecast(arr, 'uint8');
    case 'eTypeUInt16',
        arr = iDataset.GetDataVolumeAs1DArrayShorts(0, 0);
        stack(:) = typecast(arr, 'uint16');
    case 'eTypeFloat',
        stack(:) = iDataset.GetDataVolumeAs1DArrayFloats(0, 0);
    otherwise,
        error('Bad value for type');
end
```

We first have to query the type of the dataset so that we can allocate a memory block with the right type in MATLAB. We can then call the correct Ice method to get the pixel data, and then typecast the signed integer into an unsigned integer. As we will see, IceImarisConnector does everything transparently to the user with the type-agnostic method `GetDataVolume()`. Here we used the `GetDataVolumeAs1DArray<Type>` methods because it makes the transfer much faster and because the MATLAB `typecast` functions works on 1D arrays anyway.

4.3.2.2 Getting image data from MATLAB to Imaris

Copying pixel data back to Imaris **does not** require a type cast!

```
% Set the stack
switch char(iDataSet.GetType())
    case 'eTypeUInt8',
        iDataSet.SetDataVolumeAs1DArrayBytes(stack(:), 0, 0);
    case 'eTypeUInt16',
        iDataSet.SetDataVolumeAs1DArrayShorts(stack(:), 0, 0);
    case 'eTypeFloat',
        iDataSet.SetDataVolumeAs1DArrayFloats(stack(:), 0, 0);
    otherwise,
        error('Bad value for iDataSet.GetType()');
end
```

Here we can pass the stack as is, without typecast! Notice that here the usage of the `GetDataVolumeAs1DArray<Type>` methods is not strictly necessary, but it is faster. IceImarisConnector provides a `SetDataVolume()` method that takes care of all the details.

4.3.2.3 Exercise (for later)

Compare the performance of `GetDataVolumeBytes()` vs. `GetDataVolumeAs1DArrayBytes()`.

4.3.3 Objects (“proxies”) are returned as references to their base class

Whenever one asks ImarisXT to return an object, e.g. using the `GetSurpassSelection()` method[5] that returns the object that is currently selected in the Surpass Scene, ImarisXT returns a reference to the `Imaris::IDataItem` base class (see the `IDataItem` class hierarchy in figure 4.3) instead of the actual object class.

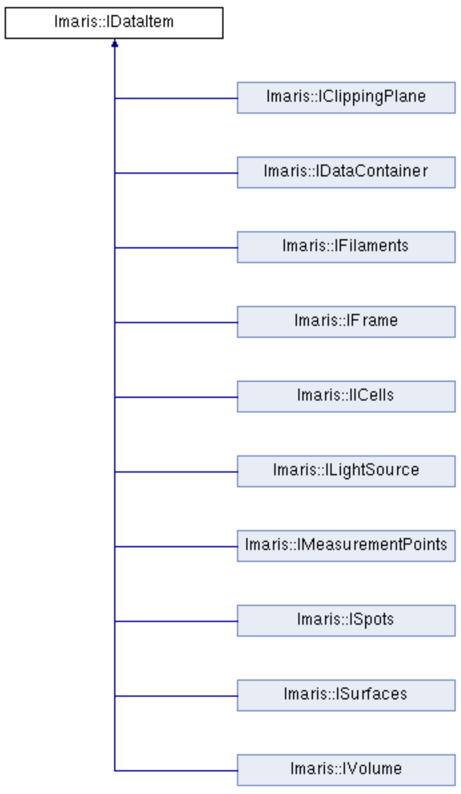


Figure 4.3: The `IDataItem` class hierarchy.

As an example, we consider the case where the selected object is an `ISpots` object, and in our MATLAB code we would like to get the spots coordinates.

```
spots = vImarisApplication.GetSurpassSelection();
if vImarisApplication.GetFactory().IsSpots(spots)
    pos = spots.GetPositionsXYZ();
end

No appropriate method, property, or field GetPositionsXYZ for
    class Imaris.IDataItemPrxHelper.
```

Notice that even though the function `isSpots()` returns **true** (i.e. the selectd object is indeed

[5]From the `IApplication` class.

an `ISpots` object), the call `spots.GetPositionsXYZ()` fails! The class of the returned object is `Imaris.IDataItemPrxHelper`, which is the Ice proxy encapsulating the `IDataItem` object we selected: `IDataItem` is the base class of `ISpots` (see figure 4.3). The `IDataItem` object does not have a `GetPositionsXYZ()` method, since this is implemented in the derived class `ISpots`. The correct way to get the positions is through a type cast:

```
spots = vImarisApplication.GetSurpassSelection();
if vImarisApplication.GetFactory().IsSpots(spots)
    spots = vImarisApplication.GetFactory().ToSpots(spots);
    pos = spots.GetPositionsXYZ();
end

spots =
    1.1454   55.3780    1.6204
    1.4394   45.3413    1.2051
   11.4817   77.0214    1.9614
   ...
```

As we will see, IceImarisConnector takes care of casting objects appropriately.

4.3.4 Row-major vs. column-major order

In computing, row-major order and column-major order describe methods for storing multidimensional arrays in linear memory. The C programming language (and all of its derivatives) use the row-major order, while MATLAB uses column-major order. Why is this important? Memory in a computer can actually be visualized as a very long 1-dimensional array of bytes. An image is a 2-dimensional array of pixels (and even three-dimensional if we consider images stacks, like the ones acquired on a confocal microscope). How can one store a 2-dimensional array in the 1-dimensional computer memory? Consider following 2x2 array:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

In row-major order, the four values are stored as 1 2 3 4. In column-major order, they are stored as 1 3 2 4. Now, Imaris is written in C++ and stores arrays in row-major order, while MATLAB uses column-major order. This means that if one transfers the following array:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

from Imaris to MATLAB, this is what MATLAB receives:

$$\begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}$$

To get the values in the same spatial organization as in Imaris, the array must be transposed in MATLAB!

4.3.5 Dimensions order

Consider also that coordinates in Imaris as stored quite naturally as $\begin{bmatrix} x & y & z \end{bmatrix}$, whereas MATLAB considers arrays as matrices, and by definition the first dimension of a matrix is the row number and the second is the column number. This boils down to mapping the Imaris coordinate $\begin{bmatrix} x & y & z \end{bmatrix}$ to the MATLAB equivalent $\begin{bmatrix} y & x & z \end{bmatrix}$. For most cases, this reduces to remembering that the first coordinate of an object position returned by Imaris is x and not y, but it can become trickier when combining positional information from objects (e.g. spots) and pixel positions in data arrays obtained from Imaris (see section 4.3.4). IceImarisConnector has a method `getDataVolumeRM()` that returns a data volume in row-major order (i.e. each plane transposed) and that practically swaps the x and y dimensions of the array. Sections 4.5 and 5.8 will hopefully make this point clearer.

4.4 Our first *XTension*: classify spots by distance

We will now write a simple *XTension* that classifies Spots by distance from some fiduciary mark, in our case the center of mass of a surface.

Start Imaris and load the `imarisXTCourse.ims` file. Change to the Surpass Scene. You should see something like figure 4.4.

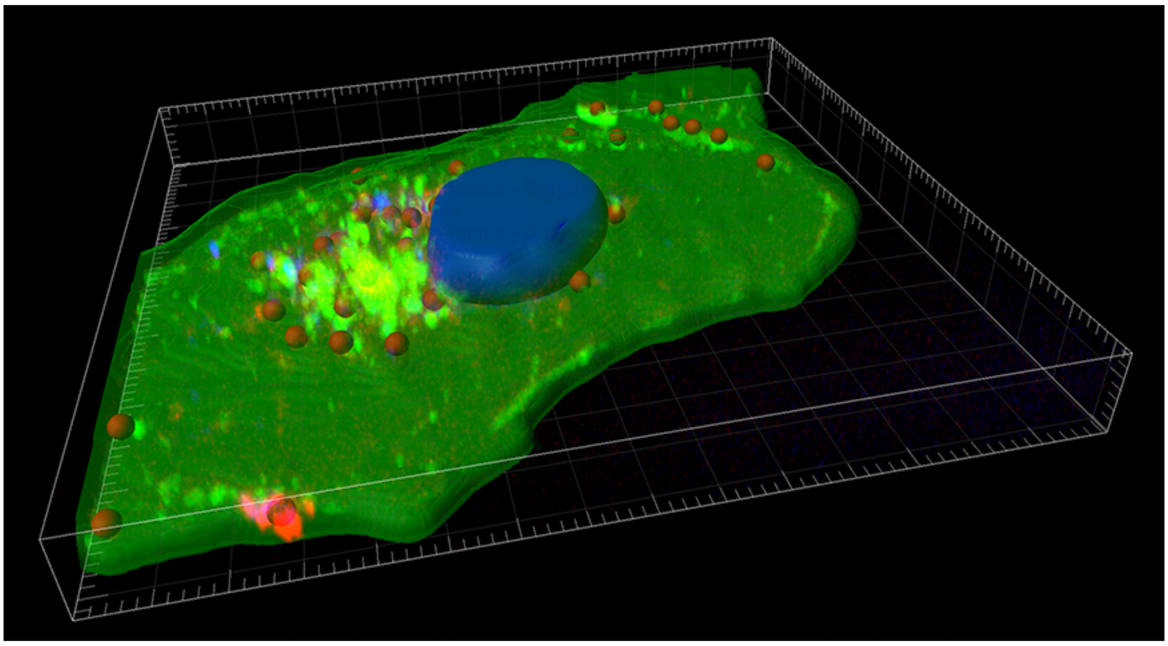


Figure 4.4: The dataset we will be working with.

To make it easier to write and test our *XTension*, we will run it from MATLAB, instead of starting it from Imaris. To get everything ready to go, perform the following steps:

- start Imaris

- open the `ImarisXTCourse.ims` file
- select the *Vesicles* Spots object
- start MATLAB
- in MATLAB, change[6] to the folder where you downloaded the *XTensions* (see 4.1)
- open the *XTension* with name *spotClassifyByDistance.m* in the editor, i.e. type:

  ```
  >> edit spotClassifyByDistance
  ```

- put a breakpoint (i.e. click on the gray gutter on the left, near the line numbers) on the first code line
- run the function as follows from the MATLAB command prompt:

  ```
  >> spotClassifyByDistance(0)
  ```

If this does not work (i.e. the Imaris Application ID is not 0), try getting it like this:

```
>> vImarisLib = ImarisLib();
>> server = vImarisLib.GetServer();
>> id = server.GetObjectID(0);
```

and then use id:

```
>> spotClassifyByDistance(id);
```

We can now execute our *XTension* line by line.

4.4.1 The *XTension* header

The exact same mechanism we saw in section 2 to add selected Fiji plug-ins to the Fiji menu is used to add MATLAB *XTensions* to the Image Processing menu. Moreover, *XTensions* can be added to the ✿ Tools tab of almost any of the Surpass Scene objects. An *XTension* is added to the menu by specifying an `<Item>` contained in a `<Menu>` and optionally one or more `<Submenu>`. The actual function to be executed is specified by the `<Command>` tag. The string `MatlabXT` must preceed the name of the MATLAB `.m` file (without extension) and the function signature is free provided the first parameter is `%i`: the ImarisApplication ID[7].

```
% Classifies the Spots by distance from the center of a surface
%
% The distance limits are specified by the user via input dialogs. The
% classified spots are copied to new Spots objects.
%
% <CustomTools>
```

[6]Type *help cd* if you need help for this.
[7]Please note that it really must be `%i`: if you use `%s` or anything else, it won't work!

```
%  <Menu>
%   <Submenu name="ImarisXT course">
%    <Item name="Classify spots by distance" icon="Matlab">
%     <Command>MatlabXT::spotClassifyByDistance(%i)</Command>
%    </Item>
%   </Submenu>
%  </Menu>
%  <SurpassTab>
%   <SurpassComponent name="bpSpots">
%    <Item name="Classify spots by distance" icon="Matlab">
%     <Command>MatlabXT::spotClassifyByDistance(%i)</Command>
%    </Item>
%   </SurpassComponent>
%  </SurpassTab>
% </CustomTools>
%
% Copyright (c) 2012, Aaron Ponti

function XTSpotsClosestDistance(aImarisApplicationID)

...
```

To add the *XTension* to the Tools tab of the Spots object, one specifies an `<Item>` contained in a `<SurpassComponent>` whose name defines the type of the object it will be assigned to. Table 4.1 lists the Surpass Component names for the various objects.

Surpass Component	Name	Remarks
Spots	bpSpots	
Surfaces	bpSurfaces	
Filaments	bpFilaments	
Cells	bpCells	
Groups (folders)	bpComponentGroup	The Surpass Scene is a bpComponentGroup
Measurement Points	bpMeasurementPoints	
Clipping Plane	bpClippingPlane	
Frame	bpFrame	

Table 4.1: List of `<SurpassComponent>` names to use in the *XTension* header.

4.4.2 The *XTension* code

This is the complete code of our *XTension*:

```
file: spotClassifyByDistance.m
```

```
function spotClassifyByDistance(aImarisApplicationID)
% Classifies the Spots by distance from the center of a surface
%
% The distance limits are specified by the user via input dialogs. The
% classified spots are copied to new Spots objects.
%
% <CustomTools>
%  <Menu>
%   <Submenu name="ImarisXT course">
%    <Item name="Classify spots by distance" icon="Matlab">
%     <Command>MatlabXT::spotClassifyByDistance(%i)</Command>
%    </Item>
%   </Submenu>
%  </Menu>
%  <SurpassTab>
%   <SurpassComponent name="bpSpots">
%    <Item name="Classify spots by distance" icon="Matlab">
%     <Command>MatlabXT::spotClassifyByDistance(%i)</Command>
%    </Item>
%   </SurpassComponent>
%  </SurpassTab>
% </CustomTools>
%
% Copyright (c) 2012, Aaron Ponti

% Set up connection between Imaris and MATLAB
if isa(aImarisApplicationID, 'Imaris.IApplicationPrxHelper')
    vImarisApplication = aImarisApplicationID;
else
    % connect to Imaris interface
    javaaddpath ImarisLib.jar
    vImarisLib = ImarisLib;
    if ischar(aImarisApplicationID)
        aImarisApplicationID = round(str2double(aImarisApplicationID));
    end
    vImarisApplication = vImarisLib.GetApplication(aImarisApplicationID);
end

% Get the currently selected object in the surpass scene and check that it
% is a Spots object
vSpots = vImarisApplication.GetSurpassSelection();
vSpots = vImarisApplication.GetFactory().ToSpots(vSpots);
if isempty(vSpots)
    errordlg('Please select a Spots object!');
    return
end

% Get the Surfaces objects and ask the user to pick one if there are more
% than one
surfaces = {}; nSurfaces = 0;
nChildren = vImarisApplication.GetSurpassScene().GetNumberOfChildren();
for i = 0 : (nChildren - 1)
    child = vImarisApplication.GetSurpassScene.GetChild( i );
    if vImarisApplication.GetFactory().IsSurfaces(child)
        nSurfaces = nSurfaces + 1;
        % We must cast the child to a Surfaces object!
        surfaces{nSurfaces} = ...
            vImarisApplication.GetFactory().ToSurfaces(child);
    end
end
if nSurfaces == 0
        errordlg('There are no Surfaces in the scene!');
    return
end
if nSurfaces > 1
    % Ask the user to pick one surface
    vSurface = askUserToPickSurface(surfaces);
    if isempty(vSurface)
        return
    end
else
    vSurface = surfaces{ 1 };
end

% Get the center of mass of the Surfaces object
centerOfMass = vSurface.GetCenterOfMass(0);

% Ask the user to specify the distance limits to use
distanceLimits = askUserToSetDistanceLimits();
if isempty(distanceLimits)
    return
end

% Get the spot coordinates, radii and time indices
spotValues = vSpots.Get();
coords = spotValues.mPositionsXYZ;
radii = spotValues.mRadii;
timeIndices = spotValues.mIndicesT;

% Calculate all spot distances from the measurement point
D = sqrt(...
    (coords(:, 1) - centerOfMass(1)).^2 + ...
    (coords(:, 2) - centerOfMass(2)).^2 + ...
    (coords(:, 3) - centerOfMass(3)).^2);

% Now classify the spots by distance and create new Spots objects
% containing the spots that fall in current distance bin
distanceLimits = [0 distanceLimits Inf];
for i = 2 : numel(distanceLimits)
    indx = find(D >= distanceLimits(i - 1) & D < distanceLimits(i));
    if ~isempty(indx)
        newSpots = vImarisApplication.GetFactory().CreateSpots();
        newSpots.Set(coords(indx, :), timeIndices(indx), radii(indx));
        newSpots.SetColorRGBA(uint32(randi(255, [1 3]) * [1; 256; 256^2]));
        newSpots.SetName(['Spots ', num2str(distanceLimits(i - 1)), ...
            ' <= D < ', num2str(distanceLimits(i)), ' [', ...
            num2str(numel(indx)), ']']);
        vImarisApplication.GetSurpassScene().AddChild(newSpots, -1);
    end
end

% ==== Helper functions ===================================================

% Ask the user to pick a surface from a list
function vSurface = askUserToPickSurface(surfaces)
nSurfaces = numel(surfaces);
surfaceNames = cell(1, nSurfaces);
for i = 1 : nSurfaces
    surfaceNames{i} = char(surfaces{i}.GetName());
end
[s, v] = listdlg( ...
    'Name', 'Question', ...
    'PromptString', ...
    'Choose surface from which the distances will be calculated', ...
    'SelectionMode', 'single', ...
    'ListSize', [400 300], ...
    'ListString', surfaceNames);
if v == 0
    vSurface = [];
    return;
end
vSurface = surfaces{ s };

% Ask the user to set the distance limits a surface from a list
% (we provide reasonable defaults)
function distanceLimits = askUserToSetDistanceLimits()
answer = inputdlg({'Please set distance limits:'}, 'Input', ...
    1, {'10, 20, 30, 40, 50'});
if isempty(answer)
    return;
end
distanceLimits = str2num(answer{1});
if isempty(distanceLimits)
    errordlg('Invalid distance limits!');
    distanceLimits = [];
    return;
end
distanceLimits = [0 distanceLimits Inf];
```

4.4.3 The *XTension* code explained

In this quick analysis of the code from previous section we will concentrate on the lines highlighted in red.

The first block of code takes care of setting up the communication between Imaris and MATLAB though Ice. This is meant to work uni-directionally, with Imaris starting MATLAB and passing its *aImarisApplicationID* (a small integer value). The code:

```
javaaddpath ImarisLib.jar
vImarisLib = ImarisLib;
```

works because when MATLAB is started, the current path is set by Imaris and is the folder where the `ImarisLib.jar` package is found. Also adding it to the Java path is fine, since `ImarisLib.jar` is not in the path at startup and MATLAB will be closed by Imaris at the end of the XTension. Running one of the bundled XTensions from MATLAB requires us to manually change the path to the folder where `ImarisLib.jar` is contained.

aImarisApplicationID can also be an IApplication object proxy (`Imaris.IApplicationPrxHelper`), but this is not really used in practice.

In the next block of code, we must explicitly cast the object returned by `GetSurpassSelection()` to an `ISpots`:

```
vSpots = vImarisApplication.GetSurpassSelection();
vSpots = vImarisApplication.GetFactory().ToSpots(vSpots);
```

otherwise we will receive an `IDataItem` instead. Please notice that if nothing is selected in the Surpass Scene or the selection object is not a spots object, `ToSpots()` will return `[]` (empty).

Next, we iterate over all children of the Surpass Scene and collect all `ISurfaces` objects we find into a cell array for further use. Again, we have to explicitly cast to `ISurfaces`.

Finally, we create `an ISpots` object containing subsets of the spots that fall within the desired range of distances from the surface center of mass. Empty `ISpots` objects are created by means of a `Factory`, are then populated with all the relevant attributes (positions, radii, time points, name, color) and then added to the Surpass Scene as a new child. Notice how the color of the Spots is set:

```
newSpots.SetColorRGBA(uint32(randi(255, [1 3]) * [1; 256; 256^2]));
```

The details of how a color is defined in Imaris can be found by typing

```
>> help IceImarisConnector/mapRgbaVectorToScalar
```

at the MATLAB prompt.

Here we only remark that the way colors are set in most of the bundled XTensions (and reproduced here) omits setting the transparency (or implicitly sets it to zero). This is because of the issue discussed in section 4.3.2 (see also section 5.7.1).

4.4.4 Exercise

As an exercise, modify the *XTension* to calculate distances from the Nucleus surface and not from its center of mass.

4.5 Our second *XTension*: plotting spots on a maximum-intensity projection

In this second *XTension*, we will calculate the maximum intensity projection (MIP) along the z axis for all channels of the `ImarisXTCourse.ims` dataset[8], and then plot the coordinates of all Spots objects found in the Surpass Scene onto the MIP.

4.5.1 The *XTension* header

In contrast to the `spotClassifyByDistance.m` function of section 4.4.2, our second *XTension* will take two input parameters (see code listing below). The first input parameter of our new `spotPlotCoordsOnDataSetMIP.m` function is the well-known Imaris application ID (`aImarisApplicationID`). But here we add a second one, `rowMajor`. As we discussed in section 4.4, when asking Imaris to return the dataset to MATLAB, it will do it in column-major order (in MATLAB) thus resulting in a dataset with swapped x and y dimensions. If we were writing this function for real usage, we would probably just transpose to row-major order to preserve the same geometry as the original dataset and give the user what he or she intuitively expects. In our example here, however, we will implement both options for the user to choose. This will allow us to see how to pass more input parameters to an *XTension* and also how to swap dimensions in a dataset. Let's start by taking a look at the function header.

```
function spotPlotCoordsOnDataSetMIP(aImarisApplicationID, rowMajor)
% Plots the z-projected Spots coordinates on the dataset MIPs
%
% To keep the code short, the function plots all found Spots objects.
% No questions asked.
%
% <CustomTools>
%  <Menu>
%   <Submenu name="ImarisXT course">
%    <Item name="Plot spot coordinates on dataset MIP (column-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 0)</Command>
%    </Item>
%    <Item name="Plot spot coordinates on dataset MIP (row-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 1)</Command>
%    </Item>
%   </Submenu>
%  </Menu>
%  <SurpassTab>
%   <SurpassComponent name="bpSpots">
%    <Item name="Plot spot coordinates on dataset MIP (column-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 0)</Command>
%    </Item>
%    <Item name="Plot spot coordinates on dataset MIP (row-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 1)</Command>
%    </Item>
%   </SurpassComponent>
%  </SurpassTab>
```

[8]However, we will only consider the first time point.

```
% </CustomTools>
%
% Copyright (c) 2012, Aaron Ponti
```

4.5.2 The *XTension* code

This is the complete code of our *XTension*:

```
file: spotPlotCoordsOnDataSetMIP.m

function spotPlotCoordsOnDataSetMIP(aImarisApplicationID, rowMajor)
% Plots the z-projected Spots coordinates on the dataset MIPs
%
% To keep the code short, the function plots all found Spots objects.
% No questions asked.
%
% <CustomTools>
%  <Menu>
%   <Submenu name="ImarisXT course">
%    <Item name="Plot spot coordinates on dataset MIP (column-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 0)</Command>
%    </Item>
%    <Item name="Plot spot coordinates on dataset MIP (row-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 1)</Command>
%    </Item>
%   </Submenu>
%  </Menu>
%  <SurpassTab>
%   <SurpassComponent name="bpSpots">
%    <Item name="Plot spot coordinates on dataset MIP (column-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 0)</Command>
%    </Item>
%    <Item name="Plot spot coordinates on dataset MIP (row-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIP(%i, 1)</Command>
%    </Item>
%   </SurpassComponent>
%  </SurpassTab>
% </CustomTools>
%
% Copyright (c) 2012, Aaron Ponti

% Set up connection between Imaris and MATLAB
if isa(aImarisApplicationID, 'Imaris.IApplicationPrxHelper')
    vImarisApplication = aImarisApplicationID;
else
    % connect to Imaris interface
    javaaddpath ImarisLib.jar
    vImarisLib = ImarisLib;
    if ischar(aImarisApplicationID)
        aImarisApplicationID = round(str2double(aImarisApplicationID));
    end
    vImarisApplication = vImarisLib.GetApplication(aImarisApplicationID);
end
```

```
% Set column-major by default
if nargin == 1
    rowMajor = 0;
end
if rowMajor ~= 0 && rowMajor ~= 1
    error(rowMajor must be wither 0 or 1.');
end

% Get the dataset
iDataSet = vImarisApplication.GetDataSet();
if isempty(iDataSet)
    errordlg('Please load a dataset!');
    return
end

% Find all Spots objects
spots = {}; nSpots = 0;
nChildren = vImarisApplication.GetSurpassScene().GetNumberOfChildren();
for i = 0 : (nChildren - 1)
    child = vImarisApplication.GetSurpassScene.GetChild( i );
    if vImarisApplication.GetFactory().IsSpots(child)
        nSpots = nSpots + 1;
        % We must cast the child to a Spots object!
        spots{nSpots} = ...
            vImarisApplication.GetFactory().ToSpots(child);
    end
end
if nSpots == 0
    errordlg('There are no Spots in the scene!');
    return
end

% Get the dataset class
switch char(iDataSet.GetType())
    case 'eTypeUInt8',   datatype = 'uint8';
    case 'eTypeUInt16',  datatype = 'uint16';
    case 'eTypeFloat',   datatype = 'single';
    otherwise,           error('Bad value for iDataSet.GetType()');
end

% To simplify, we consider at most 3 channels (and we create an RGB image
% for display)
nChannels = iDataSet.GetSizeC();
if nChannels > 3
    nChannels = 3;
end

% Size of the output image
if rowMajor == 0
    sizeX = iDataSet.GetSizeY();
    sizeY = iDataSet.GetSizeX();
else
```

```
    sizeX = iDataSet.GetSizeX();
    sizeY = iDataSet.GetSizeY();
end

% Prepare the image (either grayscale or RGB)
if nChannels == 1
    MIP = zeros([sizeY, sizeX, 1], datatype);
else
    MIP = zeros([sizeY, sizeX, 3], datatype);
end

% Get the stack for all channels (and timepoint 0).
for i = 1 : nChannels

    % Allocate memory
    stack = zeros([iDataSet.GetSizeX(), iDataSet.GetSizeY(), ...
        iDataSet.GetSizeZ()], datatype);

    % Get the pixel data
    switch char(iDataSet.GetType())
        case 'eTypeUInt8',
            arr = iDataSet.GetDataVolumeAs1DArrayBytes(i - 1, 0);
            stack(:) = typecast(arr, 'uint8');
        case 'eTypeUInt16',
            arr = iDataSet.GetDataVolumeAs1DArrayShorts(i - 1, 0);
            stack(:) = typecast(arr, 'uint16');
        case 'eTypeFloat',
            stack(:) = iDataSet.GetDataVolumeAs1DArrayFloats(i - 1, 0);
        otherwise,
            error('Bad value for type');
    end

    % Project and store
    if rowMajor == 0
        MIP(:, :, i) =  max(stack, [], 3);
    else
        MIP(:, :, i) =  (max(stack, [], 3))'; % Transposed
    end

end

% Display the MIP
figure; imshow(MIP, []);
hold on;

% Since we project along the z axis, we can ignore the z coordinates!
%
% Spot coordinates are stored in dataset units (usually µm), therefore we
% need to map them to pixel coordinates to be able to plot them on the MIP.
% For this we need the dataset voxel sizes.

% Voxel size X
voxelSizesX = (iDataSet.GetExtendMaxX() - iDataSet.GetExtendMinX()) / ...
    iDataSet.GetSizeX();
```

```
% Voxel size Y
voxelSizesY = (iDataSet.GetExtendMaxY() - iDataSet.GetExtendMinY()) / ...
    iDataSet.GetSizeY();

% Prepare some plot colors (that match our example dataset)
colors = [ 1 0 0; 0 1 0; 0 0 1];
for i = 1 : nSpots

    % Coords in dataset units
    coords = spots{i}.GetPositionsXYZ();

    % Voxels positions X
    posX = (coords(:, 1) - iDataSet.GetExtendMinX()) ./ voxelSizesX + 0.5;

    % Voxels positions Y
    posY = (coords(:, 2) - iDataSet.GetExtendMinY()) ./ voxelSizesY + 0.5;

    % Plot
    if rowMajor == 0
        plot(posY, posX, '*', 'MarkerSize', 8, 'Color', colors(i, :));
    else
        plot(posX, posY, '*', 'MarkerSize', 8, 'Color', colors(i, :));
    end

end
```

4.5.3 The *XTension* code explained

The initial part is mostly the same as in the first XTension and will not be discussed here.

The first interesting code block asks the `IDataset` object to reveal the type of the pixel data:

```
iDataSet.GetType()
```

which is one of `eTypeUInt8`, `eTypeUInt16` or `eTypeFloat` in Imaris with the corresponding `uint8`, `uint16` and `single` in MATLAB. We need to know the datatype of the dataset to allocate memory and call the correct Java methods to obtain the arrays.

Depending on whether we want the data in column- or row-major order, the MATLAB matrix that will contain the MIP will be either `[(iDataSet).GetSizeY()` by `GetSizeX()]` or `[GetSizeX()` by `GetSizeY()]`.

Since our ultimate goal is to calculate and display a MIP with the right orientation, it does not matter how we create it: in case we want it in row-major order, we just transpose the MIP after we have created it. As discussed in section 4.3.2.1, we need to typecast the byte stream that we get from Imaris to the right MATLAB data type. For performance reasons, and also because the `typecast` function in MATLAB only works on 1-dimensional arrays, we first allocate memory in the right shape:

```
stack = zeros([iDataSet.GetSizeX(), iDataSet.GetSizeY(), ...
        iDataSet.GetSizeZ()], datatype);
```

and we then fill it with a `typecast` of the obtained 1-dimensional array:

```
stack(:) = typecast(arr, 'uint8');
```

This is much faster than getting the 3D volume from Imaris already in the right shape, which is also more difficult to `typecast` properly.

ImarisXT does not offer any API to get the voxel size of the loaded dataset: it must be calculated from the dataset *extends* and the number of pixels in each dimension. This is the example for the voxel size in x direction:

```
% Voxel size X
voxelSizesX = (iDataSet.GetExtendMaxX() - iDataSet.GetExtendMinX()) / ...
    iDataSet.GetSizeX();
```

Extends, coordinates, voxel sizes, ... are all given in real-world units, most often µm. Sometimes it is useful to know on which *voxel* a particular point in space falls. This can be calculated as follows:

```
% Voxels positions X
    posX = (coords(:, 1) - iDataSet.GetExtendMinX()) ./ voxelSizesX + 0.5;
```

The `0.5` added to the position in pixels is due to the fact that an integer real wold position falls between two points in the voxel grid. To visualize this, try to picture the following: a confocal stack with `iDataSet.GetExtendMinZ` = 0 (µm) and a distance of 1 µm between planes has a z coordinate of 0.5 µm. If you look at the dataset as a series of images stacked in z direction, the plane at $z = 0.5$ um is the first image in the voxel grid.

Finally, depending on whether we transposed the MIP or not, we have to plot the spots accordingly; either:

```
plot(posY, posX, ...)
```

or

```
plot(posX, posY, ...)
```

5 Writing Imaris *XTensions* with IceImarisConnector

5.1 IceImarisConnector

IceImarisConnector is a simple commodity class written in MATLAB that eases communication between Imaris and MATLAB using the ImarisXT interface.

It wraps itself around the Ice-based Imaris Application object and provides a series of methods to facilitate several operations while ensuring direct (read-only) access to the Ice object.

5.2 Where to find

The latest stable version of IceImarisConnector can be downloaded from the project web site[1]. The git repository can be cloned from github.com[2].

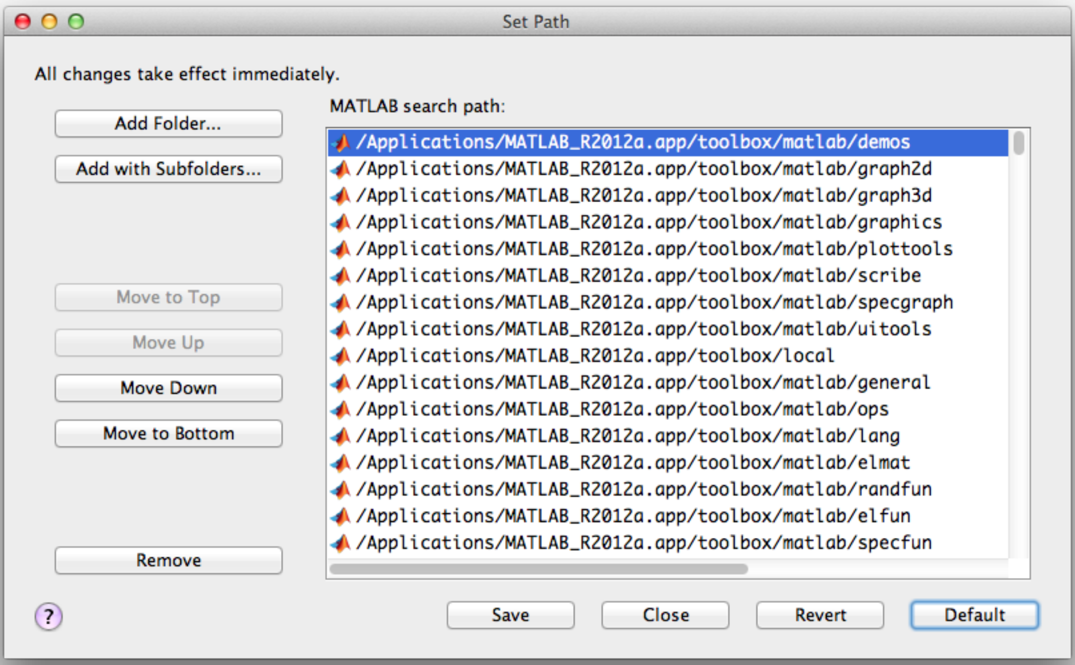
5.3 How to install

Download or clone IceImarisConnector to a folder of choice. There are several ways to use IceImarisConnector: here we will continue with the simple approach we have used so far. Copy the `@IceImarisConnector` subfolder to the same folder that contains the *XTensions* of this course and that you added to the Imaris in section 4.1. This way, Imaris will find it when running *XTensions*.

To run IceImarisConnector from MATLAB, we have to add it to the MATLAB path as well. In MATLAB, choose *Set Path* from the *File* menu, click on the *Add folder* button and choose the same directory (see figure 5.1).

To run MATLAB from IceImarisConnector, the latter must know where the former is. If Imaris is installed to the default location, IceImarisConnector should manage to find it automatically. In the case that several Imaris versions are installed, IceImarisConnector will pick the one with highest version number. If Imaris is installed in a non-standard location, or if one wants to use a version which is not the latest, the desired path can be defined by setting the environment variable IMARISPATH.

[1]http://www.scs2.net/next/index.php?id=110
[2]https://github.com/aarpon/IceImarisConnector



All changes take effect immediately.

MATLAB search path:

```
/Applications/MATLAB_R2012a.app/toolbox/matlab/demos
/Applications/MATLAB_R2012a.app/toolbox/matlab/graph2d
/Applications/MATLAB_R2012a.app/toolbox/matlab/graph3d
/Applications/MATLAB_R2012a.app/toolbox/matlab/graphics
/Applications/MATLAB_R2012a.app/toolbox/matlab/plottools
/Applications/MATLAB_R2012a.app/toolbox/matlab/scribe
/Applications/MATLAB_R2012a.app/toolbox/matlab/specgraph
/Applications/MATLAB_R2012a.app/toolbox/matlab/uitools
/Applications/MATLAB_R2012a.app/toolbox/local
/Applications/MATLAB_R2012a.app/toolbox/matlab/general
/Applications/MATLAB_R2012a.app/toolbox/matlab/ops
/Applications/MATLAB_R2012a.app/toolbox/matlab/lang
/Applications/MATLAB_R2012a.app/toolbox/matlab/elmat
/Applications/MATLAB_R2012a.app/toolbox/matlab/randfun
/Applications/MATLAB_R2012a.app/toolbox/matlab/elfun
/Applications/MATLAB_R2012a.app/toolbox/matlab/specfun
```

Figure 5.1: Add the IceImarisConnector folder to path.

5.4 Documentation

The complete API documentation can be found online[3]. Alternatively, every function is fully documented for usage in MATLAB. To get help for IceImarisConnector in MATLAB, type `help IceImarisConnector` at the MATLAB prompt:

```
>> help IceImarisConnector

  IceImarisConnector constructor

  DESCRIPTION

    IceImarisConnector is a simple commodity class that eases communication
    between Imaris and MATLAB using the ImarisXT interface.

  SYNOPSIS

    conn = IceImarisConnector(vImarisApplication, indexingStart)

  INPUT

    vImarisApplication : (optional) if omitted (or set to []), an
                         IceImarisConnector object is created that is
                         not connected to any Imaris instance.

                         Imaris can then be started (and connected) using
                         the startImaris() method, i.e.

                             conn.startImaris()

                         Alternatively, vImarisApplication can be:

                         - an Imaris Application ID as provided by Imaris
                         - an IceImarisConnector reference
                         - an Imaris Application ICE object.

    indexingStart      : (optional, default is 0) either 0 or 1, depending
                         on whether you prefer to index arrays in
                         IceImarisConnector starting at 0 or 1.

                         All indexing in ICE starts at 0; in contrast,
                         MATLAB indexing starts at 1.
                         To keep consistentcy, indexing in IceImarisConnector
                         is also 0-based (i.e. indexingStart defaults to 0).
                         This means that to get the data volume for the
                         first channel and first time point of the dataset
                         you will use conn.GetDataVolume(0, 0).
                         It you are come confortable with 1-based indexing,
                         i.e. you prefer using conn.GetDataVolume(1, 1),
                         you can set indexingStart to 1.
```

[3]http://www.scs2.net/index.php?id=113

```
                           Whatever you choose, be consistent!

    REMARK

      The Imaris Application ICE object is stored in the read-only property
      mImarisApplication. The mImarisApplication property gives access to
      the entire Imaris ICE methods. Example:

      conn.mImarisApplication.GetSurpassSelection()

      returns the currently selected object in the Imaris surpass scene.

    OUTPUT

      conn    :  an object of class IceImarisConnector
```

Help for any of the IceImarisConnector methods can be obtained like this:

```
  > help IceImarisConnector/autocast

    IceImarisConnector:  autocast (public method)

    DESCRIPTION

      This method casts IDataItems to their derived types

    SYNOPSIS

      derivedType = conn.autocast(IDataItem)

    INPUT

      IDataItem: an Imaris::IDataItem object

    OUTPUT

      derivedType : one of the Imaris::IDataItem subclasses:
                                  - Imaris::IClippingPlane
                                  - Imaris::IDataContainer
                                  - Imaris::IFilaments
                                  - Imaris::IFrame
                                  - Imaris::IDataSet
                                  - Imaris::ICells
                                  - Imaris::ILightSource
                                  - Imaris::IMeasurementPoints
                                  - Imaris::ISpots
                                  - Imaris::ISurfaces
                                  - Imaris::IVolume
                                  - Imaris::ISurpassCamera
                                  - Imaris::IImageProcessing
                                  - Imaris::IFactory
```

5.5 How to use IceImarisConnector from Imaris

In most cases, the instantiation of an IceImarisConnector object is the first action in an XT function[4]:

```
function myFancyXTension( aImarisApplicationID )
% myFancyXTension is the next great XT function
% ...

% Instantiate an IceImarisConnector object
conn = IceImarisConnector( aImarisApplicationID );

% Rest of the code...
```

Simply put, the call to the IceImarisConnector() constructor replaces the following code (which we saw in our two example *XTensions*):

```
if isa(aImarisApplicationID, 'Imaris.IApplicationPrxHelper')
    vImarisApplication = aImarisApplicationID;
else
    % connect to Imaris interface
    javaaddpath ImarisLib.jar
    vImarisLib = ImarisLib;
    if ischar(aImarisApplicationID)
        aImarisApplicationID = round(str2double(aImarisApplicationID));
    end
    vImarisApplication = vImarisLib.GetApplication(aImarisApplicationID);
end
```

In reality, it does quite a bit more, since it takes care of adding ImarisLib.jar to the path *only once*, it makes sure the Ice server is running before a connection is attempted, and more[5].

The *aImarisApplicationID* input parameter is the well-know ID that Imaris passes to the *XTension* when it is launched. IceImarisConnector uses it to initialize the connection to Imaris. The `conn` object returned by the constructor is our powerful IceImarisConnector object, which gives us access to the Imaris Application object (what we called `vImarisApplication` in our first *XTensions*) and adds several additional functionalities via the IceImarisConnector API[6]. See the next section for more details.

5.6 How to use IceImarisConnector from MATLAB

Using IceImarisConnector from MATLAB allows for a better experience, especially while writing and debugging new *XTensions*, since MATLAB can be used with full functionality. Moreover, once the connection between MATLAB and Imaris is established, it is maintained until Imaris is closed by the user (or via IceImarisConnector), meaning that neither MATLAB nor Imaris must be restarted before an *XTension* can be re-run.

An IceImarisConnector object can be instantiated without parameters:

[4]Not considering parameter checking.
[5]Take a look at the code: it's open source!
[6]http://www.scs2.net/next/index.php?id=113

```
>> conn = IceImarisConnector()
IceImarisConnector: not connected to an Imaris instance yet.
```

In case you did not set the IMARISPATH environment variable, you can check what IceImarisConnector found by using the `info()` method:

```
>> conn.info()
IceImarisConnector version 0.3.3 using:
- Imaris path: /Applications/Imaris 7.7.1.app
- Imaris executable: /Applications/Imaris 7.7.1.app/Contents/MacOS/Imaris
- ImarisServer executable: /Applications/Imaris 7.7.1.app/Contents/MacOS/ImarisServerIce
- ImarisLib.jar archive: /Applications/Imaris 7.7.1.app/Contents/SharedSupport/XT/matlab/ImarisLib.jar
```

Now the IceImarisConnector object can be used to start an Imaris instance and connect to it.

```
>> conn.startImaris();
```

MATLAB and Imaris are now connected through the IceImarisConnector object. The Imaris Application Ice object is a read-only property (member) of the class and can be accessed with the usual dot notation:

```
>> conn.mImarisApplication
ans =

    67093A5B-72AD-4206-A6B8-D51CDD078634 -t:tcp -h xxx.xxx.xxx.xxx -p 50183
```

If an Imaris application is already running, one can link it to IceImarisConnector by passing the ID 0 to the IceImarisConnector constructor[7]:

```
>> conn = IceImarisConnector(0)
IceImarisConnector: connected to Imaris.
```

Importantly, the IceImarisConnector constructor can also take a reference to an IceImarisConnector object as a parameter, so that our fancy *XTension* from section 5.5 can be called also as follows:

```
>> myFancyXTension(conn);
```

since the `conn` object will be passed on to the IceImarisConnector constructor at the beginning ot the *XTension*. If the `conn` object is storing an active connection, it will be re-used.

5.7 Our first *XTension* revised: classify spots by distance using IceImarisConnector

This is the *XTension* from section 4.4.2 rewritten using IceImarisConnector:

[7] See also section 4.4.

file: spotClassifyByDistanceIIC.m

```
function spotClassifyByDistanceIIC(aImarisApplicationID)
% Classifies the Spots by distance from the center of a surface using IceImarisConnector
%
% The distance limits are specified by the user via input dialogs. The
% classified spots are copied to new Spots objects.
%
% <CustomTools>
%  <Menu>
%   <Submenu name="ImarisXT course">
%    <Item name="Classify spots by distance using IIC" icon="Matlab">
%     <Command>MatlabXT::spotClassifyByDistanceIIC(%i)</Command>
%    </Item>
%   </Submenu>
%  </Menu>
%  <SurpassTab>
%   <SurpassComponent name="bpSpots">
%    <Item name="Classify spots by distance using IIC" icon="Matlab">
%     <Command>MatlabXT::spotClassifyByDistanceIIC(%i)</Command>
%    </Item>
%   </SurpassComponent>
%  </SurpassTab>
% </CustomTools>
%
% Copyright (c) 2012, Aaron Ponti

% Set up connection between Imaris and MATLAB
conn = IceImarisConnector(aImarisApplicationID);

% Get the currently selected object in the surpass scene
vSpots = conn.getSurpassSelection();
if isempty(vSpots)
  errordlg('Please select a Spots object!');
  return
end

% Get the Surfaces objects
surfaces = conn.getAllSurpassChildren(0, 'Surfaces');
nSurfaces = numel(surfaces);
if nSurfaces == 0
  errordlg('There are no Surfaces in the scene!');
  return
end
if nSurfaces > 1
  % Ask the user to pick one surface
  vSurface = askUserToPickSurface(surfaces);
  if isempty(vSurface)
    return
  end
else
  vSurface = surfaces{ 1 };
end
```

```
% Get the center of mass of the Surfaces object
centerOfMass = vSurface.GetCenterOfMass(0);

% Ask the user to specify the distance limits to use
distanceLimits = askUserToSetDistanceLimits();
if isempty(distanceLimits)
  return
end

% Get the spot coordinates, radii and time indices
spotValues = vSpots.Get();
coords = spotValues.mPositionsXYZ;
radii = spotValues.mRadii;
timeIndices = spotValues.mIndicesT;

% Calculate all spot distances from the measurement point
D = sqrt(...
    (coords(:, 1) - centerOfMass(1)).^2 + ...
    (coords(:, 2) - centerOfMass(2)).^2 + ...
    (coords(:, 3) - centerOfMass(3)).^2);

% Now classify the spots by distance and create new Spots objects
% containing the spots that fall in current distance bin
for i = 2 : numel(distanceLimits)
  indx = find(D >= distanceLimits(i - 1) & D < distanceLimits(i));
  if ~isempty(indx)
    name = ['Spots ', num2str(distanceLimits(i - 1)), ...
        ' <= D < ', num2str(distanceLimits(i)), ' [', ...
        num2str(numel(indx)), ']'];
    conn.createAndSetSpots(coords(indx, :), timeIndices(indx), ...
        radii(indx), name, [rand(1, 3) 0]);
  end
end

% ==== Helper functions ==================================================
... (identical to those in the code from section 4.4.2)
```

5.7.1 The *XTension* code explained

The XTension obviously delivers the same result as the one in section 4.4 but does it in quite fewer lines.

The responibility for setting up (and maintaining) the connection between Imaris and MATLAB is taken by the IceImarisConnector constructor, which returns an IceImarisConnector object `conn` ready for us to use.

The call:

```
vSpots = conn.getSurpassSelection();
```

returns a `ISpots` object already casted to the derived class (see section 4.3.3).

```
surfaces = conn.getAllSurpassChildren(0, 'Surfaces');
```

returns all children of the Surpass Scene that are surfaces already casted to the correct class `ISurfaces`.

All the steps required to create and add `ISpots` objects to the Surpass scene are encapsulated in the call:

```
conn.createAndSetSpots(coords(indx, :), timeIndices(indx), ...
        radii(indx), name, [rand(1, 3) 0]);
```

Please notice that the last input parameter of `createAndSetSpots()`[8] is the RGBA color of the spots, which is an 1-by-4 vector containing the R, G, B, and A values (between 0 and 1) that represent the Red, Green, and Blue color components and the transparency value of the `ISpots` object. Behind the scenes, `createAndSetSpots()` calls `mapRgbaVectorToScalar()` that takes care of converting the RGBA vector into the unsigned 32-bit integer that Imaris uses to set the color and transparency of the `ISpots` object working around the signed-integer issue we discussed in section 4.3.2[9].

5.8 Our second *XTension* revised: plotting spots on a maximum-intensity projection using IceImarisConnector

This is the *XTension* from section 4.5.2 rewritten using IceImarisConnector:

```
file: spotPlotCoordsOnDataSetMIPIIC.m

function spotPlotCoordsOnDataSetMIPIIC(aImarisApplicationID, rowMajor)
% Plots the z-projected Spots coordinates on the dataset MIPs using IceImarisConnector
%
% To keep the code short, the function plots all found Spots objects.
% No questions asked.
%
% <CustomTools>
%  <Menu>
%   <Submenu name="ImarisXT course">
%    <Item name="Plot spot coordinates on dataset MIP using IIC (column-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIPIIC(%i, 0)</Command>
%    </Item>
%    <Item name="Plot spot coordinates on dataset MIP using IIC (row-major)" icon="Matlab">
%     <Command>MatlabXT::spotPlotCoordsOnDataSetMIPIIC(%i, 1)</Command>
%    </Item>
%   </Submenu>
%  </Menu>
%  <SurpassTab>
%   <SurpassComponent name="bpSpots">
%    <Item name="Plot spot coordinates on dataset MIP using IIC (column-major)" icon="Matlab">
```

[8]The full signature of `createAndSetSpots` is: `createAndSetSpots(coords, timeIndices, radii, name, color)`.

[9]Interestingly, the problem occurs only when trying to set a transparency value different than zero (the reason why is explained in the help of `IceImarisConnector/mapRgbaVectorToScalar`). Indeed, none of the bundled Imaris *XTensions* ever changes the transparency.

```
%      <Command>MatlabXT::spotPlotCoordsOnDataSetMIPIIC(%i, 0)</Command>
%     </Item>
%     <Item name="Plot spot coordinates on dataset MIP using IIC (row-major)" icon="Matlab">
%      <Command>MatlabXT::spotPlotCoordsOnDataSetMIPIIC(%i, 1)</Command>
%     </Item>
%    </SurpassComponent>
%   </SurpassTab>
%  </CustomTools>
%
% Copyright (c) 2012, Aaron Ponti

% Set up connection between Imaris and MATLAB
conn = IceImarisConnector(aImarisApplicationID);

% Set column-major by default
if nargin == 1
    rowMajor = 0;
end
if rowMajor ~= 0 && rowMajor ~= 1
    error(rowMajor must be wither 0 or 1.');
end

% Get the dataset sizes := [sizeX sizeY sizeZ sizeC sizeT]
[sizeX sizeY sizeZ sizeC sizeT] = conn.getSizes();
% Get the dataset
if sizeX == 0
    errordlg('Please load a dataset!');
    return
end

% Find all Spots objects
spots = conn.getAllSurpassChildren(1, 'Spots');
nSpots = numel(spots);
if nSpots == 0
    errordlg('There are no Spots in the scene!');
    return
end

% Get the datatype
datatype = conn.getMatlabDatatype();

% Our display will be either grayscale (for one channel), or RGB, for any
% other number of channels
if sizeC == 1
    nChannels = 1;
else
    nChannels = 3;
end

% Prepare the image with the correct size (either grayscale or RGB)
if rowMajor == 0
    MIP = zeros([sizeX, sizeY, nChannels], datatype);
else
    MIP = zeros([sizeY, sizeX, nChannels], datatype);
```

```
end

% Get the stack for all channels (and timepoint 0).
for i = 1 : sizeC

    if i > nChannels
        break;
    end

    % Get the stack
    if rowMajor == 0
        stack = conn.getDataVolume(i - 1, 0);
    else
        stack = conn.getDataVolumeRM(i - 1, 0);
    end

    % Project and store
    MIP(:, :, i) =  max(stack, [], 3);

end

% Display the MIP
figure; imshow(MIP, []);
hold on;

% Since we project along the z axis, we can ignore the z coordinates!
%
% Spot coordinates are stored in dataset units (usually um), therefore we
% need to map them to pixel coordinates to be able to plot them on the MIP.
% Prepare some plot colors (that match our example dataset)
colors = [ 1 0 0; 0 1 0; 0 0 1];
for i = 1 : nSpots

    % Coords in dataset units
    coords = spots{i}.GetPositionsXYZ();

    % Map the coordinates to pixels
    [posX, posY] = conn.mapPositionsUnitsToVoxels(coords);

    % Plot
    if rowMajor == 0
        plot(posY, posX, '*', 'MarkerSize', 8, 'Color', colors(i, :));
    else
        plot(posX, posY, '*', 'MarkerSize', 8, 'Color', colors(i, :));
    end

end
```

5.8.1 The *XTension* code explained

IceImarisConnector provides a few commodity function to get sizes (`getSizes()`), extends (`getExtends()`), voxel sizes (`getVoxelSizes()`), type of the Imaris dataset mapped to the MATLAB type (`getMatlabDatatype()`) and also mapping functions to convert from real world units to voxel (`mapPositionsUnitsToVoxels()`) and from voxels to units (`mapPositionsVoxelsToUnits()`).

Getting data volumes from Imaris both in column-major and row-major order is straightforward as well: `getDataVolume()` returns the 3-Dimensional dataset for a given channel and time point in column-major order, whereas `getDataVolumeRM()` returns it in row-major order.

5.8.2 Exercise

Match the colors of the plotted MIP channels and spots to those in the Imaris Surpass Scene.