Science & Technology Facilities Council

Technical Report
RAL-TR-2013-007

Developing an image analysis plugin for VolView

DJ Worth

June 2013

ISSN 1358- 6254

Developing an Image Analysis Plugin for VolView

D.J. Worth

May 2013

Abstract

VolView is a multiplatform volume visualisation application from KitWare built on VTK and ITK and allowing user plugins to extend its functionality. As such it was chosen by the UK Collaborative Computational Project in Tomographic Imaging (CCPi) as the platform for its image analysis algorithms. This report describes the development of a plugin to VolView. The basic format for a plugin is given in the VolView User Guide and there is an extensive collection of plugins which also provide guidance. However there are no details of how a plugin should be compiled and we rectify that situation in this report and hope that others will now be able to build plugins more efficiently.

Keywords: CCPi, VolView, volume visualisation, image analysis, C++, plugin

Email: `david.worth@stfc.ac.uk`
Reports can be obtained from: `http://epubs.stfc.ac.uk`

Software Engineering Group
Scientific Computing Department
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

The UK Collaborative Computational Project on Tomographic Imaging (CCPi) had a number of algorithms in image quantification which they wanted to make more widely available to the image community. As part of the core support for this CCP from the Scientific Computing Department at STFC these algorithms were to be packaged for volume visualisation software using good software engineering processes as well as taking into consideration the efficiency of the implementation. The initial implementations were in a wide variety of languages including, C, Matlab and IDL. Some implementations targeted specific display software while others were simply used for a "one-off" analysis task but the potential for wider application has been recognised.

The majority of the algorithms had a Matlab implementation and they could have been left as that however there are costs associated with buying the Matlab software and there is no readily available interface for manipulating and analysing the image: e.g. cleaning the image by filtering, measuring components, altering transparency of certain intensities. These criteria lead us to consider alternative visualisation software and following a study of a range of applications VolView was chosen as the preferred platform. It offered a wide range of existing functionality for manipulation and analysis, it could read raw volume data and could be extended with user plugins in C or C++ which could make use of the Visualisation Toolkit (VTK – `http://www.vtk.org/`) and the Insight Segmentation and Registration Toolkit (ITK – `http://www.itk.org/`) which provide an extensive library of algorithms for image analysis. Another important factor is that VolView is available as open source software giving the possibility of developing the application itself.

Whilst VolView was the preferred platform we acknowledge that one of the leading commercial packages is Avizo from the Visualization Services Group (`http://www.vsg3d.com/avizo/overview`). Avizo also makes use of ITK and allows users to write their own plugins. In order to keep our options open to target Avizo at some later time the design of the plugin described here must separate the infrastructure for communication between the application and the plugin, from the plugin itself. Similarly for testing purposes it will be convenient to run the plugin from a simple command-line application and this must be borne in mind for the plugin design. In fact the test application could be a useful tool in its own right.

The source code and pre-compiled VolView application and shared libraries that implement this plugin can be obtained from the Image Quantification Algorithms project on CCPForge `http://ccpforge.cse.rl.ac.uk/gf/project/iqa/`

2 Anatomy of a VolView Plugin

The entirety of the information on VolView plugins is contained in the user manual [1] and in the source code for the plugins distributed with the VolView source code. Together they do a good job in describing how to write a plugin, whether in C or C++ and possibly making use of VTK and/or ITK for reading files and performing calculations. This section gives a brief overview of the structure of a plugin and leaves the details to the user manual.

An API is provided as a set of C functions that the developer must implement and which are called by VolView as it interacts with the plugin and C data structures which are used for exporting and importing data and transferring GUI parameters from the VolView application. Figure 1 (taken from the user manual) shows how VolView learns about and interacts with the plugin. The source code has to implement the 3 functions shown from `vtkVVPluginAPI.h` paying particular attention to the naming of the initialisation function. The code should then be compiled to a shared library for the particular operating system and that shared library file placed in the `Plugins` subdirectory of the directory where the VolView application is installed.

There is a strong link between the name of the plugin, the initialisation (`Init`) function name and the name of the shared library or DLL file that VolView loads. For example a plugin named

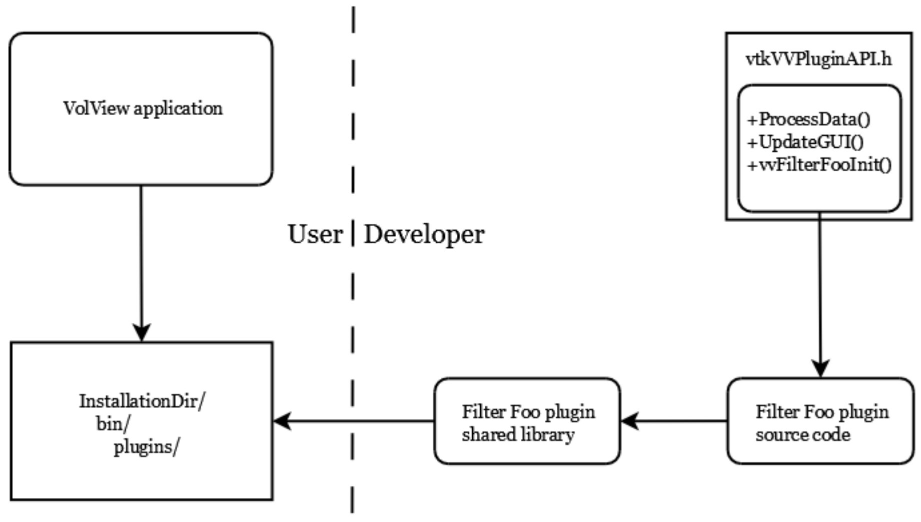

Figure 1: The interaction between VolView application and plugin

vvFooFilter the `Init` function must be named `vvFooFilterInit()` and the shared library must be named `libvvFooFilter.so` in Unix/Linux/OsX and `vvFooFilter.dll` in Windows.

2.1 The Initialisation Function

This function is called as the VolView application starts before the user interface is displayed. It performs the following actions:

- Assigns the functions for `ProcessData` and `UpdateGUI`.
- Defines the place in the plugin menu structure where this plugin is found.
- Gives documentation for the plugin that is shown in VolView.
- Defines other properties of the plugin that help VolView call the plugin efficiently.

The function must conform to the following API

```
extern "C"
{
  void VV_PLUGIN_EXPORT vvEllipsoidFittingInit(vtkVVPluginInfo *info)
  { }
}
```

and it is the `info` data structure that is used to implement the actions in the list above. The symbol `VV_PLUGIN_EXPORT` and the data structure `vtkVVPluginInfo` are both defined in the `vtkVVPluginAPI.h` header file. This header file contains a lot of useful information on types of parameters, datatypes, GUI components, and parameter values that can be used when implementing a plugin.

The source code for the initialisation function for the quantification plugin can be found in Appendix A.

2.2 Defining the Plugin's User Interface

VolView allows the user to programmatically create the user interface for the plugin as part of the plugin code. There are a limited number of GUI elements that can be used but they allow for simple parameterisation of the algorithms. We will describe them later in this section.

The signature for the function that VolView will call is as follows

```
static int UpdateGUI(void *inf)
{}
```

The first argument is a pointer to a `vtkVVPluginInfo` data structure which can be recovered by a simple cast as follows:

```
vtkVVPluginInfo *info = (vtkVVPluginInfo *)inf;
```

The function `SetGUIProperty()` can be called on this data structure to define the GUI elements. It also contains the output data set information which should also be set – most importantly the output scalar data type. Often this is the same as the data type of the input volume but it can be any of the other supported data types listed in Table 1.

Type	**Macro**	**Type**	**Macro**
void	`VTK_VOID`	bit	`VTK_BIT`
char	`VTK_CHAR`	unsigned char	`VTK_UNSIGNED_CHAR`
short	`VTK_SHORT`	unsigned short	`VTK_UNSIGNED_SHORT`
int	`VTK_INT`	unsigned int	`VTK_UNSIGNED_INT`
long	`VTK_LONG`	unsigned long	`VTK_UNSIGNED_LONG`
float	`VTK_FLOAT`	double	`VTK_DOUBLE`
id	`VTK_ID_TYPE`		

Table 1: The datatypes used in VolView

We will now describe the three types of interface elements available for a plugin. The source code for this function for the quantification plugin is given in Appendix B.

Number value with scale

There are 4 components of this element plus alternative ways to set the range for the value. All are set with the `SetGUIProperty()` function on the `vtkVVPluginInfo` data structure. The table below describes each component and the macro used to identify it.

Description	**Macro**	**Notes**
Type of element	`VVP_GUI_TYPE`	Use `VVP_GUI_SCALE`
Text label	`VVP_GUI_LABEL`	A `char*`
Default value	`VVP_GUI_DEFAULT`	A `char*` with the value
Help text	`VVP_GUI_HELP`	A `char*`. Pops up when mouse hovers over the element. *Make good use of this.*

Table 2: Defining a numerical value with scale

There are two alternative methods of defining the limits for the scale. The first is to use the limits defined in the language for the C data type of the input data:

```
vvPluginSetGUIScaleRange(element_id);
```

where `element_id` is 0 for the first defined element, 1 for the second and so on.

The second method which is best for values that have nothing to do with the input data is to give a minimum, maximum and step size in a single string as follows:

```
info->SetGUIProperty(info, element_id, VVP_GUI_HINTS , "1 1000 1");
```

Drop down list

The same 4 components are used for this element with the `VVP_GUI_TYPE` set to `VVP_GUI_CHOICE` and the default set to one of the list entries defined with the `VVP_GUI_HINTS` macro.

To illustrate how to define the entries in the list take the example of the ImageMathematics plugin which allows the user to perform a simple arithmetical operation on the voxel values of two images. The operations are addition, subtraction, multiplication, absolute difference and division and the symbols are concatenated in a string separated by `\n`. The number of entries, in this case 5, is given at the start of the string. The line of code to define the list is therefore:

```
info->SetGUIProperty(info, 0, VVP_GUI_HINTS, "5\n+\n-\n*\n|-|\n/");
```

Checkbox

Again the same 4 components are used for this element with the `VVP_GUI_TYPE` set to `VVP_GUI_CHECKBOX`. The value of the check box is either 0 – not checked; or 1 – checked.

2.3 The ProcessData Function

This is the function in which the work is done. It can call other functions, can make use of other classes written by the user and the VTK and ITK libraries. The `info` data from the initialisation function is passed to it along with a data structure containing details of the data set to be processed. The signature for the function is

```
static int ProcessData(void *inf, vtkVVProcessDataStruct *pds)
{ }
```

The first argument is a pointer to a `vtkVVPluginInfo` and can be accessed in the same way as for setting up the user interface described above. This data structure contains the following data that are useful for processing the data:

`InputVolumeScalarType` The VTK enumeration of the data type of the input data. This is useful when operations work differently on different data types or for VTK/ITK classes are used which are templated on the data type.

`InputVolumeDimensions` Number of voxels in each dimension

`InputVolumeSpacing` Can be thought of as the "size" of the voxel in each dimension. Useful with VTK/ITK classes.

`InputVolumeOrigin` Origin position for the dataset. Useful with VTK/ITK classes.

These data are for the input volume and there are corresponding variables for a possible second input data set and the output data set. Those for the output data set should be set by the plugin.

The second argument, a data structure holding information on the volume data, is defined in the `vtkVVPluginAPI.h` header file. Particular parts of the data structure that are of most interest are:

`inData` Pointer to the input data

`outData` Pointer to the output data. Already correctly sized to the same dimension as the input volume. The data type must be set in the `UpdateGUI` function which is called when the plugin is selected in the application. Outputs of the processing are returned to the application in this data.

`inData2` Pointer to second input data for a plugin that uses 2 input data sets.

Values from the user interface can be retried from the `info` data structure. They are returned as `char*` data can be used this way to detect the chosen entry from a drop down list or the standard C functions `atoi` and `atof` can be used to get the numerical value. An example in which we get the value of element 0, is

```
const unsigned int numberOfIterations =
    atoi( info->GetGUIProperty(info, 0, VVP_GUI_VALUE ) );
```

The plugin described in this document is written in C++ and the standard VolView C++ plugin implementation uses a template function according to the input volume data type. This function is called with a VolView supplied macro function. The source code of the ProcessData and the template function are given in Appendix C

3 Quantification of a Labelled Image

The quantification described here starts from a *labelled* image, that is one in which all the voxels have been given a value that allocates them to a particular connected feature in the image or to the background (zero value). Every voxel in a connected feature has the same label. An example image is given in Figure 2

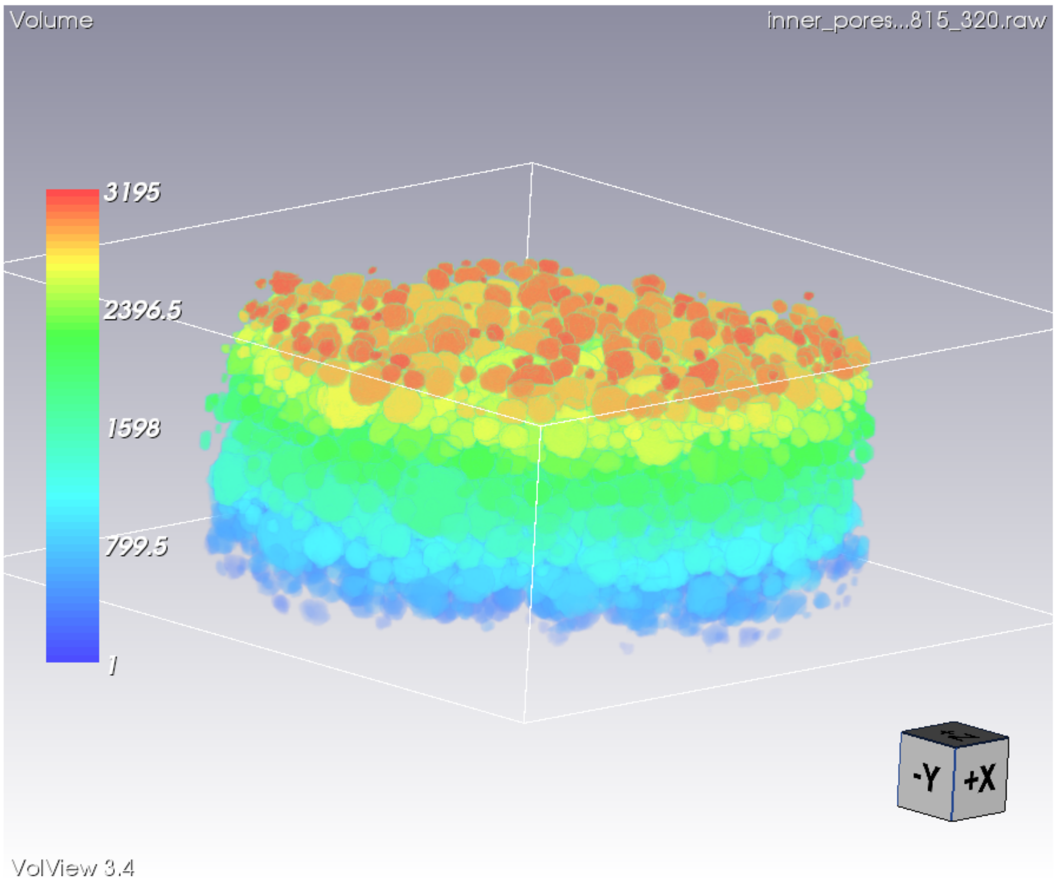

Figure 2: An example labelled image

Quantification of this image will give an idea of the size, shape and orientation of each of the features from which statistical distributions can be extracted. One application is in the manufacture of synthetic bone grafts (scaffolds) used to aid regeneration of diseased or damaged bone. These scaffolds should have particular structural properties to be effective and also degrade as host bone forms. A non-destructive method is therefore required for quantifying the pore and interconnect sizes and the degradation of the scaffold over time. The quantification method implemented here has been successfully used for this task – see [2] for details.

The plugin calculates the following characteristics for each feature:

- Volume by voxel count
- Equivalent sphere diameter by voxel count. Use the formula $2(3V/(4\pi))^{1/3}$ where V is the volume by voxel count.
- Bounding box diagonal
- Principal Component Analysis. Eigenanalysis of covariance matrix of voxel positions in this feature. Eigenvectors give local coordinate axes for the feature and eigenvalues give length of feature along respective axis.
- Ellipsoid fitting by PCA to calculate external surface of feature.
- Equivalent circle diameter by PCA to find largest cross-sectional area of feature.
- Isosurface. Another way of calculating the external surface of the feature.
- From the isosurface: the surface area, volume enclosed, equivalent sphere diameter of the volume.
- Sphericity of isosurface. $\pi^{1/3}(6V)^{2/3}/A$ where V is the volume enclosed by the isosurface and A is its area.
- Normalised surface area to volume ratio. rA/V where r is the equivalent sphere diameter of the volume.

For each feature the set of calculations is the same and there is no overlap of data between features or requirement of one feature's data prior to any other. This means we can do the calculations in parallel with no data sharing.

4 Plugin Design

We obviously have to follow the API for VolView plugins but once this is achieved we have free reign on the implementation of the data processing. Having identified that the calculations can be performed in parallel for each feature in the volume we should also bear this in mind in the design. It is most likely that VolView will be used on a desktop machine which we can assume will have multiple cores and so we should make use of openMP to schedule the calculations for each feature.

The first step will be to scan through the volume data and store the voxels belonging to each feature. Then for each feature we take the set of voxels and perform the calculations. There will be a great deal of supporting data that is required during the quantification for each feature and this data will be of varying sizes. For this reason we will use an object to store this data and perform the calculations for each feature as it comes under consideration. This object will be called **`CCPiQuantificationWorker`**. We will also have an object to scan the volume data and store the voxels by feature and perform any functions required in the overall volume context. This volume context object will be able to create the object for each feature when required. It will be called **`CCPiQuantification`**. The recipe for using these classes is as follows

An object diagram for this design is given in Figure 4.

```
Create CCPiQuantification object
Initialise this object from VolView data
Create the list of voxels for each labelled feature
Write results header
For each feature
        Create the CCPiQuantificationWorker object
        Run the worker object
        Write results
```

Figure 3: Pseudo-code for driving the quantification of each feature

5 Plugin Implementation

Given the object oriented design described above implementation of the plugin is in C++ which allows us to use VTK and ITK classes most naturally when they are required. We can also make use of standard template types such as `vector<T>` if they are more suitable. In this section we will describe the driver function that runs the calculations and implementation details of the two classes in the design.

5.1 The Driver Function

This is a straightforward implementation of the algorithm in Figure 3. The details to highlight are the signature of the function and the use of openMP to schedule the calculation for each feature in parallel and the implications of this for interaction with VolView.

In Section 2.3 we noted that the `ProcessData` function calls a function templated on the input volume data type via a VolView supplied macro function. The macro function matches the given function name and the number of arguments (3 in this case) and calls the template function with the arguments which means that function signature must be

```
template <class IT>
void vvQuan3DTemplate(vtkVVPluginInfo *info,
                      vtkVVProcessDataStruct *pds,
                      IT *)
```

where `info` and `pds` have been described elsewhere and the third argument is only used to provide the template datatype.

The template function can now use the two quantification classes to carry out the work. The first part is to prepare the `CCPiQuantification` class with the data from VolView, print some summary data to the console and the header for the CSV results file; and the second is to schedule the calculations for each feature.

Progress is reported to the user as a fraction of the task that has been completed (second argument) via the `info` data structure as follows:

```
info->UpdateProgress(info,0.01,"Initialising...");
```

The text given as the third argument is printed in the status bar of VolView to report what is going on next.

The quantification calculation for each feature is completely separate from any other and so we can schedule the next calculation to run as soon as a thread becomes free. An alternative would be to divide up the calculations into blocks for each thread but this will be inefficient because some features may take much longer than others to process. We use an openMP directive to loop over the features[1] and carry out *dynamic* scheduling with

[1]Of which there are `totalVoxels` number

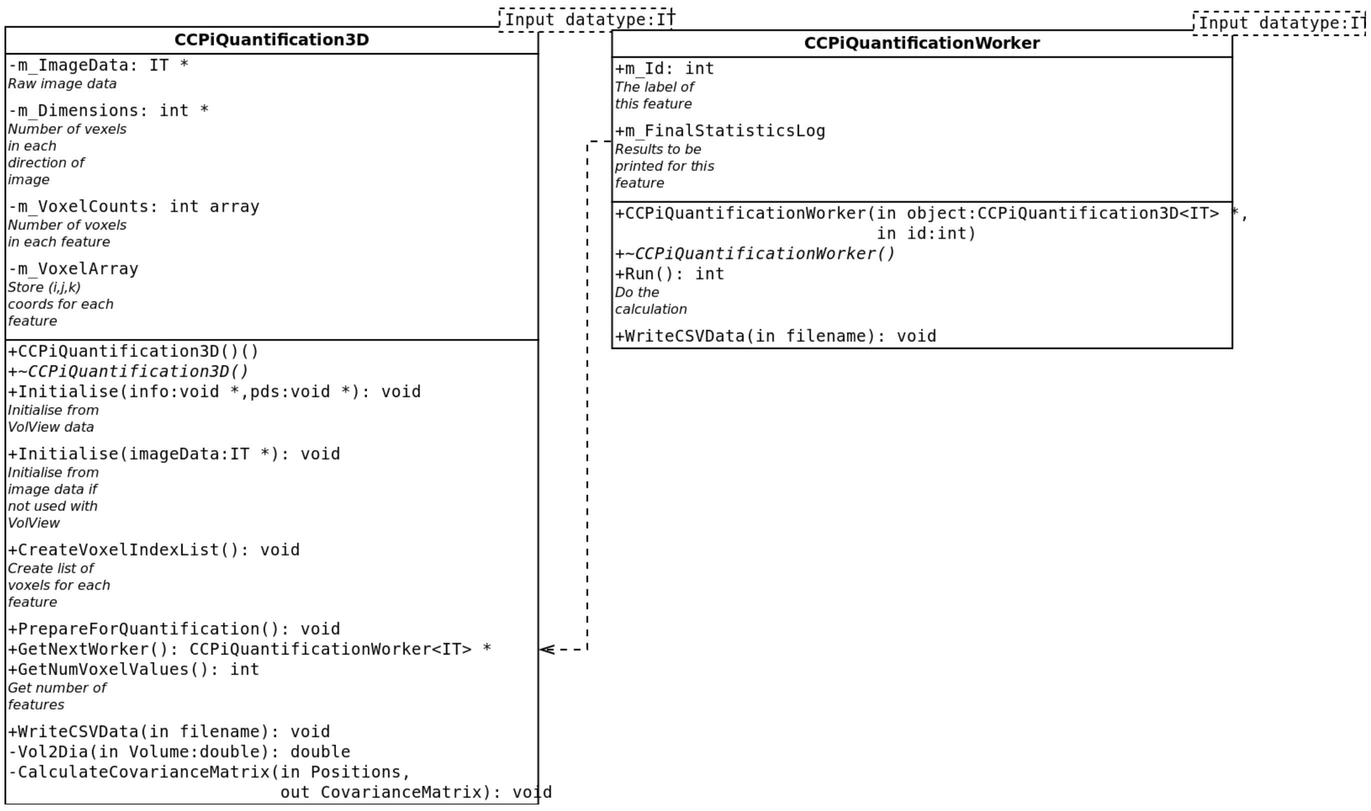
Input datatype:IT
CCPiQuantification3D
-m_ImageData: IT *
Raw image data
-m_Dimensions: int *
Number of vexels in each direction of image
-m_VoxelCounts: int array
Number of voxels in each feature
-m_VoxelArray
Store (i,j,k) coords for each feature
+CCPiQuantification3D()()
+~CCPiQuantification3D()
+Initialise(info:void *,pds:void *): void
Initialise from VolView data
+Initialise(imageData:IT *): void
Initialise from image data if not used with VolView
+CreateVoxelIndexList(): void
Create list of voxels for each feature
+PrepareForQuantification(): void
+GetNextWorker(): CCPiQuantificationWorker<IT> *
+GetNumVoxelValues(): int
Get number of features
+WriteCSVData(in filename): void
-Vol2Dia(in Volume:double): double
-CalculateCovarianceMatrix(in Positions, out CovarianceMatrix): void
Input datatype:I
CCPiQuantificationWorker
+m_Id: int
The label of this feature
+m_FinalStatisticsLog
Results to be printed for this feature
+CCPiQuantificationWorker(in object:CCPiQuantification3D<IT> *, in id:int)
+~CCPiQuantificationWorker()
+Run(): int
Do the calculation
+WriteCSVData(in filename): void

Figure 4: The object diagram for classes that process the data

```
#pragma omp parallel for schedule(dynamic)
for(int i = 0; i < totalVoxels; i++) {
```

Each feature is quantified by a worker object created by the `CCPiQuantification` class. This introduces a slight complication into the parallelisation as we must ensure that each thread gets a separate worker which requires that the creation is done within an openMP *critical* block. Workers can run in parallel but to prevent results becoming jumbled when workers print to the output file we must use another critical block for the writing of the results.

Unfortunately there is one (seemingly) insurmountable problem for us when using openMP. We cannot use the function to update the progress in VolView or the function to check whether the user has cancelled the running of the plugin. Therefore we have implemented a simple text based progress meter that can be seen when running VolView from the command line. Again we must use a critical block to make sure that the appearance of the progress meter is correct.

The source code of the driver function is given in Appendix D

5.2 Details for CCPiQuantification

This class stores data relating to the volume as a whole which it receives from the VolView `vtkVVPluginInfo` and `vtkVVProcessDataStruct` data structures. It keeps a pointer to the volume data as well as the number of components in the image, the VTK data type of the input, the number of voxels in each direction and the origin and spacing of the data. These last two data are required by VTK classes used by the worker when calculating the largest cross-sectional area of the feature and the isosurface.

The class also provides some utility functions for the worker class to calculate the diameter of a sphere from its volume, to find the minimum, maximum and mean positions of a set of points and to calculate the covariance matrix from a set of points.

The data types of many of the variables and arguments to functions are determined by which methods act on those variables. For example, the covariance matrix is a 3×3 double precision ITK matrix because the eigenanalysis is performed by an ITK class. The eigenvectors are stored in a similar matrix and using this matrix to rotate the voxel positions requires the positions be stored as an other ITK matrix.

To avoid transferring data relating to the input volume we make the worker class a friend class of `CCPiQuantification`.

5.3 Details for CCPiQuantificationWorker

The details about the volume as a whole are kept by the `CCPiQuantification` class to which the `CCPiQuantificationWorker` class keeps a pointer. The worker class uses its `m_Id` member variable to access data relating to its allocated feature.

The worker class has the sole purpose of doing the quantification for a single feature. Users call the `Run` method to start the calculation and the return value is 0 if the calculation was run or 0 if the feature does not have enough voxels to be significant.

We use the following VTK classes to help with the quantification:

`vtkPointsProjectedHull` To calculate the largest cross-sectional area of the feature. The voxel positions are rotated to have the largest PCA derived axes in the x and y directions before inserting the points into the class. The `GetCCWHullZ` method calculates the hull projected onto the $x - y$ plane and `vtkPolygon::ComputeArea` can be used to calculate the area.

`vtkSynchronizedTemplates3D` To calculate the isosurface with input from `VTKImageImport` and output to .

`vtkMassProperties` To calculate the surface area and volume enclosed. This takes a `vtkPolyData` object which represents the isosurface (output from `vtkSynchronizedTemplates3D`) as its input.

6 Build System

The build system for a VolView plugin is a hand crafted *Makefile* coupled with the build directory and source for VolView itself. The VolView build directory contains all the libraries for VTK and ITK and the source directory contains the header file `vtkVVPluginAPI.h` which defines the API for a VolView plugin and also header files for VTK and ITK.

It is necessary to build VolView before building a plugin and this is done using the CMake utility `http://www.cmake.org/` as follows:

```
%> wget http://www.kitware.com/VolView/files/VolViewSrc.tar.gz
%> tar zxvf VolViewSrc.tar.gz
%> cd VolViewSrc
%> mkdir build
%> cd build
%> cmake ../VolViewPlatformComplete/
%>....Install missing dependencies identified by cmake...
%> cmake ../VolViewPlatformComplete/
%> make
```

The library files to link against will be in the `bin` subdirectory of the `build` directory created above and the include files for VTK and ITK under `VolViewExternalLibraries` directory in `VTK` and `Insight` respectively.

It is up to the developer to search for the header files and libraries and include the appropriate directories and files in the include search paths for compilation and library search paths and library specification for linking. The Makefile for the quantification plugin is given in Appendix E.

A useful command on Linux/Unix to find the library that contains an unresolved symbol (`vtkImageImport::New()` in this case) in the linking stage is

```
%>  nm -A -C *.so | grep "vtkImageImport::New()"
```

The `nm` command lists symbols from object files with the `-A` flag to print the file name for each symbol and `-C` to demangle the names of C++ objects. This is piped to `grep` which find the lines with the required symbol. The output of this command will look something like the following and the library we require is the one with `T` in the column before the symbol name – `libvtkImaging.so` in this case.

```
libKWVolView.so:                 U vtkImageImport::New()
libvtkImaging.so:0000000000243fda T vtkImageImport::New()
libvtkImagingTCL.so:                 U vtkImageImport::New()
libvtkKWEWidgets.so:                 U vtkImageImport::New()
libvvVTKCheckerBoard.so:                 U vtkImageImport::New()
libvvVTKDilate3D.so:                 U vtkImageImport::New()
libvvVTKErode3D.so:                 U vtkImageImport::New()
libvvVTKGradientMagnitude.so:                 U vtkImageImport::New()
libvvVTKImageCast.so:                 U vtkImageImport::New()
libvvVTKMedian.so:                 U vtkImageImport::New()
libvvVTKMergeTets.so:                 U vtkImageImport::New()
libvvVTKResample.so:                 U vtkImageImport::New()
libvvVTKShrink.so:                 U vtkImageImport::New()
libvvVTKSmooth.so:                 U vtkImageImport::New()
```

References

[1] VolView 3.2 User Manual, June 23, 2009.

[2] Evaluation of 3-D bioactive glass scaffolds dissolution in a perfusion flow system with X-ray microtomography, S Yue, PD Lee, G Poologasundarampillai, JR Jones, *Acta biomaterialia 7 (2011)*, 2637-2643

A The Initialisation Function

```
/**
\brief Initialise this plugin.

Sets the name and group for this plugin in the plugin list shown to the VolView
user and gives some documentation. Also defines properties so VolView can judge
the memory requirements and potential for undoing this plugin.

\param info Pointer to object that should be modified to give details about this
plugin.
*/
extern "C"
{
  void VV_PLUGIN_EXPORT vvQuan3DInit(vtkVVPluginInfo *info)
  {
    /* Always check the version */
    vvPluginVersionCheck();

    /* Set up information that never changes */
    info->ProcessData = ProcessData;
    info->UpdateGUI = UpdateGUI;

    /* Set the properties this plugin uses */
    info->SetProperty(info, VVP_NAME, "Quantify 3D");
    info->SetProperty(info, VVP_GROUP, "Quantification");

    /* Set terse and full documentation displayed to user */
    info->SetProperty(info, VVP_TERSE_DOCUMENTATION,
     "Quantify several characteristics from a labelled image");
    info->SetProperty(info, VVP_FULL_DOCUMENTATION,
      "Quantify the following characteristics from a labelled image: Volume by \
voxel counts, Equivalent sphere diameter by voxel counts, Bounding box \
diagonal, Principal Component Analysis, Ellipsoid fitting by PCA, Equivalent \
circle diameter by PCA, Isosurface by marching cube, Surface area, Surface \
volume, Equivalent sphere diameter from surface volume, Sphercity, Normalised \
surface area to volume ratio. \nPLEASE NOTE that the Cancel button and progress \
bar do not work for this OpenMP plug-in.");

    /* Set these two values to "0" or "1" based on how your plugin
     * handles data all possible combinations of 0 and 1 are valid. */
    info->SetProperty(info, VVP_SUPPORTS_IN_PLACE_PROCESSING, "1");
    info->SetProperty(info, VVP_SUPPORTS_PROCESSING_PIECES,   "0");

    /* Set the number of GUI items used by this plugin */
    info->SetProperty(info, VVP_NUMBER_OF_GUI_ITEMS,          "1");

    info->SetProperty(info, VVP_REQUIRED_Z_OVERLAP,           "0");
    info->SetProperty(info, VVP_REQUIRES_SERIES_INPUT,        "0");
    info->SetProperty(info, VVP_SUPPORTS_PROCESSING_SERIES_BY_VOLUMES, "0");
    info->SetProperty(info, VVP_PRODUCES_OUTPUT_SERIES, "0");
    info->SetProperty(info, VVP_PRODUCES_PLOTTING_OUTPUT, "0");
  }
}
```

B Defining the Plugin User Interface

```
/**
\brief Update the VolView GUI to display user parameters.

Sets one GUI parameter - the physical size of the voxels in the image. It
gets called prior to the plugin executing.

\param inf Pointer to object that should be modified to set up GUI elements for
this plugin. It also contains details of the input and output images.
*/
static int UpdateGUI(void *inf)
{
  vtkVVPluginInfo *info = (vtkVVPluginInfo *)inf;

  /* Create required GUI elements here */
  info->SetGUIProperty(info, 0, VVP_GUI_LABEL, "Minimum feature Size");
  info->SetGUIProperty(info, 0, VVP_GUI_TYPE, VVP_GUI_SCALE);
  info->SetGUIProperty(info, 0, VVP_GUI_DEFAULT , "100");
  info->SetGUIProperty(info, 0, VVP_GUI_HELP,
               "Minimum number of voxels for a feature to be analysed");

  /* Range for possible output values */
  info->SetGUIProperty(info, 0, VVP_GUI_HINTS , "1 1000 1");


  /* By default the output image's properties match those of the input */
  info->OutputVolumeScalarType = info->InputVolumeScalarType;
  info->OutputVolumeNumberOfComponents = info->InputVolumeNumberOfComponents;
  for (int i = 0; i < 3; i++) {
    info->OutputVolumeDimensions[i] = info->InputVolumeDimensions[i];
    info->OutputVolumeSpacing[i] = info->InputVolumeSpacing[i];
    info->OutputVolumeOrigin[i] = info->InputVolumeOrigin[i];
    }

  return 1;
}
```

C The ProcessData Function

```
/**
\brief Function called by VolView to run the plug in.

It hands off the actual work to the template function vvQuan3DTemplate.

\param inf Pointer to object that holds data for this plugin. The voxel size
can be obtained from it.
\param pds Pointer to object that carries information on data set to be
processed. Includes actual buffer of voxel data, number of voxels along each
dimension, voxel spacing and voxel type.
*/
static int ProcessData(void *inf, vtkVVProcessDataStruct *pds)
{
  vtkVVPluginInfo *info = (vtkVVPluginInfo *)inf;

  switch (info->InputVolumeScalarType) {
    vtkTemplateMacro3(vvQuan3DTemplate, info, pds,
                      static_cast<VTK_TT *>(0));
  }
  return 0;
}

/**
\brief Template function to calculate the data from the image

\c IT represents the input volumes data type (e.g. float, short, etc).

\author David Worth, STFC
\date May 2012

\param info Pointer to object that holds data for this plugin. The voxel size
can be obtained from it.
\param pds Pointer to object that carries information on data set to be
processed. Includes actual buffer of voxel data, number of voxels along each
dimension, voxel spacing and voxel type.
*/
template <class IT>
void vvQuan3DTemplate(vtkVVPluginInfo *info,
                      vtkVVProcessDataStruct *pds,
                      IT *)
```

D The Driver Function

```
/**
\brief Template function to calculate the data from the image

\c IT represents the input volumes data type (e.g. float, short, etc).

\author David Worth, STFC
\date May 2012

\param info Pointer to object that holds data for this plugin. The voxel size
can be obtained from it.
\param pds Pointer to object that carries information on data set to be
processed. Includes actual buffer of voxel data, number of voxels along each
dimension, voxel spacing and voxel type.
*/
template <class IT>
void vvQuan3DTemplate(vtkVVPluginInfo *info,
                      vtkVVProcessDataStruct *pds,
                      IT *)
{
  CCPiQuantification3D<IT> quan3D;

  info->UpdateProgress(info,0.01,"Initialising...");

  quan3D.Initialise((void*)info, (void*)pds);

  quan3D.CreateVoxelIndexList();

  quan3D.PrepareForQuantification();

  quan3D.PrintSummaryData();

  quan3D.WriteCSVData("3D_data.csv");

  int totalVoxels = quan3D.GetNumVoxelValues(), n = 0;

  info->UpdateProgress(info,0.05,"Processing...");

  #pragma omp parallel for schedule(dynamic)
  for(int i = 0; i < totalVoxels; i++) {

    CCPiQuantificationWorker<IT> *worker = NULL;

    // Do the real work
    #pragma omp critical(nextworker)
    {
      worker = quan3D.GetNextWorker();
    }
    if (worker != NULL) {

      if (0 == worker->Run()) {
        #pragma omp critical(writefile)
        {
          worker->WriteCSVData("3D_data.csv");
        }
      }
      delete worker;
    }
    #pragma omp atomic
    n++;
    #pragma omp critical(progress)
    {
      int count = 40*n/totalVoxels;
      std::cout << '\r';
      for (int j = 1; j < count+1; j++) {
        std::cout << '*';
      }
      for (int j = count+1; j < 41; j++) {
        std::cout << '.';
      }
      std::cout << "|    " << 100*n/totalVoxels << "%";
    }
  }
  std::cout << std::endl;

  info->UpdateProgress(info,(float)1.0,"Processing Complete");
}
```

E Makefile

```
# Makefile for VolView plugins
#
# David Worth STFC May 2012

CC=g++

VOLVIEW=/home/djw/CCPi/VolViewSrc

VV_INCLUDE=-I$(VOLVIEW)/VolViewLibraries/KWVolView/Plugins

ITK_INCLUDE=-I$(VOLVIEW)/VolViewExternalLibraries/Insight/Code/Common \
-I$(VOLVIEW)/VolViewPlatformComplete/build/ITK

VXL_INCLUDE=-I$(VOLVIEW)/VolViewExternalLibraries/Insight/Utilities/vxl/core \
-I$(VOLVIEW)/VolViewExternalLibraries/Insight/Utilities/vxl/vcl \
-I$(VOLVIEW)/VolViewPlatformComplete/build/ITK/Utilities/vxl/vcl \
-I$(VOLVIEW)/VolViewPlatformComplete/build/ITK/Utilities/vxl/core

VTK_INCLUDE=-I$(VOLVIEW)/VolViewExternalLibraries/VTK/Graphics \
-I$(VOLVIEW)/VolViewExternalLibraries/VTK/Common \
-I$(VOLVIEW)/VolViewExternalLibraries/VTK/Filtering \
-I$(VOLVIEW)/VolViewExternalLibraries/VTK/Imaging \
-I$(VOLVIEW)/VolViewPlatformComplete/build/VTK

APP_INCLUDE=-I$(VOLVIEW)/VolViewExternalLibraries/VTK/IO/ \
-I$(VOLVIEW)/VolViewExternalLibraries/Insight/Code/IO \
$(VTK_INCLUDE) $(ITK_INCLUDE) $(VXL_INCLUDE) $(VV_INCLUDE)


CFLAGS=$(VV_INCLUDE) $(ITK_INCLUDE) $(VXL_INCLUDE) $(VTK_INCLUDE) -fPIC -Wall -std=c++98 -g -fopenmp
# Flags for compiler optimisation
#CFLAGS=$(VV_INCLUDE) $(ITK_INCLUDE) $(VXL_INCLUDE) $(VTK_INCLUDE) -fPIC -Wall -std=c++98 \
 -O3 -D_FORTIFY_SOURCE=0

LIB_DIR=$(VOLVIEW)/VolViewPlatformComplete/build/bin

LDFLAGS=-L$(LIB_DIR) $(LIB_DIR)/libITKIO.so.3.2.0 \
$(LIB_DIR)/libITKCommon.so.3.2.0 $(LIB_DIR)/libITKBasicFilters.so.3.2.0 \
$(LIB_DIR)/libITKAlgorithms.so.3.2.0 $(LIB_DIR)/libITKNrrdIO.so.3.2.0 \
$(LIB_DIR)/libitkjpeg12.so.3.2.0 $(LIB_DIR)/libitkjpeg16.so.3.2.0 \
$(LIB_DIR)/libgdcm.so $(LIB_DIR)/libgdcmjpeg8.so $(LIB_DIR)/libgdcmjpeg12.so \
$(LIB_DIR)/libgdcmjpeg16.so $(LIB_DIR)/libvtkopenjpeg.so.1.3.0 \
$(LIB_DIR)/libvtkopenjp3dvm.so $(LIB_DIR)/libitkpng.so.3.2.0 \
$(LIB_DIR)/libitktiff.so.3.2.0 $(LIB_DIR)/libitkjpeg8.so.3.2.0 \
$(LIB_DIR)/libITKSpatialObject.so.3.2.0 $(LIB_DIR)/libITKMetaIO.so.3.2.0 \
$(LIB_DIR)/libitkopenjpeg.so.1.3.0 $(LIB_DIR)/libitkopenjp3dvm.so \
$(LIB_DIR)/libITKDICOMParser.so.3.2.0 $(LIB_DIR)/libITKEXPAT.so.3.2.0 \
$(LIB_DIR)/libITKniftiio.so.3.2.0 $(LIB_DIR)/libITKznz.so.3.2.0 \
$(LIB_DIR)/libitkzlib.so.3.2.0 $(LIB_DIR)/libITKNumerics.so.3.2.0 \
$(LIB_DIR)/libITKStatistics.so.3.2.0 $(LIB_DIR)/libitkvnl_inst.so.3.2.0 \
$(LIB_DIR)/libitkvnl_algo.so.3.2.0 $(LIB_DIR)/libitkvnl.so.3.2.0 \
$(LIB_DIR)/libitkv3p_netlib.so.3.2.0 $(LIB_DIR)/libitkvcl.so.3.2.0 \
$(LIB_DIR)/libitksys.so.3.2.0 -lpthread -ldl -lm

APP_LDFLAGS=-L$(LIB_DIR) $(LIB_DIR)/libvtkFiltering.so $(LIB_DIR)/libvtkCommon.so \
$(LIB_DIR)/libvtkGraphics.so $(LIB_DIR)/libvtkImaging.so $(LIB_DIR)/libitkvnl.so \
$(LIB_DIR)/libITKCommon.so $(LIB_DIR)/libvtkIO.so

vvQuan3D.so : Quan3D.o vvQuan3D.o QuanWorker.o
$(CC) -fPIC -Wall -shared -Wl,-soname,$@ -o $@ $^ $(LDFLAGS) -fopenmp

# Use option below in line above to get linker to tell you what symbols are
# missing rather than find out at run time
# -Wl,--no-undefined

vvQuan3D.o : vvQuan3D.cpp Quan3D.hpp QuanWorker.hpp
$(CC) $(CFLAGS) -c $<

Quan3D.o : Quan3D.cpp
$(CC) $(CFLAGS) -c $<

Quan3D.cpp : Quan3D.hpp QuanWorker.hpp

QuanWorker.o : QuanWorker.cpp
$(CC) $(CFLAGS) -c $<

QuanWorker.cpp : QuanWorker.hpp

app: AppQuan3D.o Quan3D.o QuanWorker.o
$(CC) -o $@ $^ $(APP_LDFLAGS) -fopenmp

AppQuan3D.o: AppQuan3D.cpp Quan3D.hpp QuanWorker.hpp
$(CC) -Wall -std=c++98 -g $(APP_INCLUDE) -c $< -fopenmp

clean :
rm -f *.o *.so*
```