Report on the development of prototype software (Task 3 D6.3)

NMI-3 Work-package 6 FP7/NMI3-II project number 283883
Aug 29th, 2014 - R. Leal and E. Farhi (with input from members of the work-package)

Version 0.1

Abstract

This report documents the Task 3 of the Data Analysis Standards work-package (NMI3-II/WP6). It details the software that was produced during this project, with code for Mantid and other projects.

Table of Contents

Introduction....2
Contributed Mantid loaders....2
Contributed Mantid algorithms....3
Contributed Mantid instrument definitions....4
Contributed Mantid framework changes....4
Other contributions: AllToMantid and reductionSever....5
AllToMantid....5
reductionServer....5
Appendix: code produced....5
LoadILL....6
LoadILLAscii....12
LoadILLIndirect....16
LoadILLReflectometry....20
LoadILLSANS....25
LoadLLB....31
LoadSINQFocus → LoadSINQ....35
ConvertEmptyToTof....38
CorrectFlightPaths....43
DetectorEfficiencyCorUser....45
SaveILLCosmosAscii....48
SetupILLD33Reduction....49
AllToMantid: communicator....57
AllToMantid: workspace....59
AllToMantid: lamp....61
ReductionServer: main....63

Introduction

In the scope of this Data Analysis work-package, we have implemented a number of methods to analyse data from recent instruments. The aim is to demonstrate that, given possibly large data sets, we are able to import experimental data, visualise, correct and finally analyse it.

As already reported in our Task 1 and Task 2 documents, the Mantid project fulfils most of the recommendations from our Reports, and this project was then chosen as primary prototype development environment. However, a few other satellite projects were also tested.

Easy Mantid items were completed first (continuous neutron source ToF and SANS raw import). The case of the 'moving' instruments was then studied, but the initial solution making use of groups of workspaces to hold scanned/iterative acquisitions did not work properly. Effort was then redirected towards SANS reduction, back-scattering (indirect), and other developments (*AllToMantid, McStas import, reductionServer*). Actually, a usable, but not ideal, solution was found for some scanning instruments (e.g. continuous neutron source powder diffractometers) by storing data sets into a single multi-dimensional workspace. Finally, effort was devoted to writing example scripts to be used for ToF, and SANS continuous neutron source based instruments.

One of the key points which motivated the use of Mantid was the ability to re-use the *(q,ω)* 4D data reduction routines, such as VATES (derived from Horace <http://horace.isis.rl.ac.uk/>). The instruments which can benefit from these Mantid algorithms are TAS and large detector ToF spectrometers. Progress has been made in the support of continuous neutron source-based ToF spectrometers, but unfortunately there was not enough time to test VATES. We encourage the instrument responsibles of these spectrometers to validate this methodology and check that it can effectively be applied with their data sets.

In total, we have contributed to the Mantid project with more than 200 commits, and about 17 000 lines of code (LOC). Two other prototypes have been contributed in order to allow the Mantid project to make use of other external contributions, without effectively coding Mantid algorithms. Since facilities are increasingly making use of the NeXus format for data storage, all loaders (except one) developed in this work-package focus on that standard. Instruments t ILL, LLB and PSI have been included. The extension to other facilities/instruments should be facilitated when they adopt NeXus.

These contributions allow to test Mantid with a reduced but comprehensive set of continuous neutron source instruments, e.g. for ToF, Back-Scattering, SANS, Reflectometer, and partially diffractometer.

All the code produced should be considered as a prototype. Most code is associated with test procedure, as well as example data files. Other Python scripts have been written to generate the IDF for new instruments (ILL, LLB, SINQ) and visualise data sets independently of Mantid.

Some of this code is listed in Appendix to this report, and corresponds to state on July 16th 2014. Our Mantid contributions have been included in the project, and should be maintained by the Mantid developers in the future. The Mantid software can be obtained at <www.mantidproject.org>.

Contributed Mantid loaders

These algorithms take as input e.g. a file name, and produce a Mantid workspace in memory, usually a 2D one, except for D2B which produces a MD workspace.

Algorithm	Description	Instrument
LoadILL	Loads a ILL nexus file.	ILL: IN4, IN5 and IN6
LoadILLAscii	Loads ILL Raw data in Ascii format.	ILL: D2B
LoadILLIndirect	Loads a ILL/IN16B nexus file.	ILL: IN16B
LoadILLReflectometry	Loads a ILL/D17 nexus file.	ILL: D17
LoadILLSANS	Loads a ILL nexus files for SANS instruments.	ILL: D33
LoadLLB	Loads LLB nexus file.	LLB: MiBemol
LoadSINQFocus → LoadSINQ	Loads a FOCUS nexus file from the PSI	SINQ: FOCUS

Other algorithms have been developed in parallel with this work-package, by other contributors: LoadMcStas, LoadMcStasNeXus, LoadAscii.

Contributed Mantid algorithms

These algorithms take as input one or more workspaces, and correct them to produce new workspaces. The data corrections are usually rather basic, except for a few cases.

Algorithm	Description	Class
CalculateEfficiency	Calculates the detector efficiency for a SANS instrument.	SANS
ConvertEmptyToTof	Converts the channel number to time of flight.	TOF (ILL)
CorrectFlightPaths	Used to correct flight paths in 2D shaped detectors.	TOF (ILL: IN5)
DetectorEfficiencyCorUser	This algorithm calculates the detector efficiency according the formula set in the instrument definition file/parameters.	TOF (ILL: IN4, IN5, IN6)
EQSANSDarkCurrentSubtraction	Perform EQSANS dark current subtraction.	SANS (minor modification)
EQSANSQ2D	Workflow algorithm to process a reduced EQSANS workspace and produce I(Qx,Qy).	SANS (minor modification)
SANSAzimuthalAverage1D	Compute I(q) for reduced SANS data	SANS (minor modification)
SANSBeamFinder	Beam finder workflow algorithm for SANS instruments.	SANS (minor modification)

Algorithm	Description	Class
SANSSensitivityCorrection	Perform SANS sensitivity correction.	SANS (minor modification)
SaveILLCosmosAscii	Saves a 2D workspace to a ascii file usable by COSMOS/LAMP	SANS
SetupILLD33Reduction	Set up ILL D33 SANS reduction options.	SANS (ILL: D33)
TransmissionUtils		SANS (minor modification)
IDF_to_PLY	Convert an IDF into PLY/OFF	Geometry (prototype)

Contributed Mantid instrument definitions

The instrument definitions specify the instrument geometry (IDF), e.g. the detector elements w.r.t. the sample position. This geometry information is used both for visualisation purposes, as well as for the direct to reciprocal space transformation. The definitions can be attached to parameters, which can be changed when reading the experimental data, especially for movable parts.

File	Description	Has configurable parameters
D2B_Definition	ILL: D2B	
D17_Definition	ILL: D17	Yes
D33_Definition	ILL: D33	Yes
FOCUS_Definition	SINQ: FOCUS	
IN4_Definition	ILL: IN4	Yes
IN5_Definition	ILL: IN5	Yes
IN16_Definition	ILL: IN16	Yes
IN16B_Definition	ILL: IN16B	Yes
MIBEMOL_Definition	LLB: MiBemol	

We have also produced a few Python scripts to generate most of these IDF. In addition, definitions for IN10 and IN13 have been contributed independently.

The instrument definitions are not listed in the Appendix as they may be large.

Contributed Mantid framework changes

As stated previously, the Mantid infrastructure is currently not adapted to describe scanning instruments which produce a concatenated data set, such as when using a Triple-Axis Spectrometer, or a scanning powder diffractometer. In order to overcome this limitation, a prototype upgrade of the geometry layer in Mantid was tested.

- Geometry/src/Instrument.cpp
- Geometry/inc/MantidGeometry/Instrument/ParameterMap.h
- Geometry/src/Instrument/CompAssembly.cpp

The idea is to be able, from set of separate workspaces each attached to a single, static, instrument geometry, to define a virtual instrument definition which is composed of an array of geometries. This new implementation was tested by merging two separate D2B acquisition scan steps into a virtual instrument workspace. However, this low level framework modification had impact on the whole Mantid project, and generated many issues with existing algorithms. It was thus discarded. It was estimated that about a man year is necessary to properly implement this functionality.

Other contributions: AllToMantid and reductionSever

AllToMantid

This Mantid Algorithm takes as input a Mantid workspace, exports it into a NeXus file, launches an external software which should produce as a result a treated NeXus file, which is then imported back into Mantid. The mechanism uses a pipe to communicate with the external software, and can be adapted to any external software. To date, it has been tested with LAMP <http://www.ill.eu/?id=3463> and iFit <http://ifit.mccode.org>.

The full code of this project can be obtained at <https://github.com/ricleal/AllToMantid>.

reductionServer

The reductionServer is an ILL REST Live data reduction server.

The purpose of this project is to bridge data acquisition and data analysis. This server seats in the middle of the instrument control computer and the data reduction and analysis software.

The instrument control computer initiate the data analysis requests by sending a JSON message to the server. (e.g. via *curl*) The server reacts to these requests and forward the respective demands to the data analysis software. The server implements a Representational State Transfer (REST) with messages passed in JSON format. When the data analysis software has finished the processing, the end status is communicated back to the requester, with the result. Intermediate messages can also me generated to estimate the progress of a computation.

The code of this project can be obtained at <https://github.com/ricleal/reductionServer>.

Appendix: code produced

The full produced code is available at the work-package web site <http://nmi3.eu/about-nmi3/networking/data-analysis-standards.html>. As Mantid is a rapidly evolving project, the code below is not guaranteed to work in the future, and is thus only given as example.

LoadILL	Loads a ILL nexus file.	ILL: IN4, IN5 and IN6

```
//----------------------------------------------------
// Includes
//----------------------------------------------------
#include "MantidDataHandling/LoadILL.h"
#include "MantidAPI/FileProperty.h"
#include "MantidAPI/Progress.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidGeometry/Instrument.h"
#include "MantidKernel/EmptyValues.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidDataHandling/LoadHelper.h"

#include <boost/algorithm/string/predicate.hpp> // boost::starts_with

#include <limits>
#include <algorithm>
#include <iostream>
#include <vector>
#include <cmath>

namespace Mantid
{
  namespace DataHandling
  {

    using namespace Kernel;
    using namespace API;
    using namespace NeXus;

    DECLARE_NEXUS_FILELOADER_ALGORITHM(LoadILL);

    /**
     * tostring operator to print the contents of NXClassInfo
     *
     * TODO : This has to go somewhere else
     */
    std::ostream& operator<<(std::ostream &strm, const NXClassInfo &c)
    {
      return strm << "NXClassInfo :: nxname: " << c.nxname << " , nxclass: " << c.nxclass;
    }

//----------------------------------------------------
// Private member functions
//----------------------------------------------------

    /**
     * Return the confidence with with this algorithm can load the file
     * @param descriptor A descriptor for the file
     * @returns An integer specifying the confidence level. 0 indicates it will not be used
     */
    int LoadILL::confidence(Kernel::NexusDescriptor & descriptor) const
    {

      // fields existent only at the ILL
      if (descriptor.pathExists("/entry0/wavelength")
          && descriptor.pathExists("/entry0/experiment_identifier")
          && descriptor.pathExists("/entry0/mode")
          && !descriptor.pathExists("/entry0/dataSD") // This one is for LoadILLIndirect
          && !descriptor.pathExists("/entry0/instrument/VirtualChopper") // This one is for LoadILLReflectometry
              )
      {
        return 80;
      }
      else
      {
        return 0;
      }
    }

    LoadILL::LoadILL() :
        API::IFileLoader<Kernel::NexusDescriptor>()
    {
      m_instrumentName = "";
      m_wavelength = 0;
      m_channelWidth = 0;
      m_numberOfChannels = 0;
      m_numberOfHistograms = 0;
      m_numberOfTubes = 0;
      m_numberOfPixelsPerTube = 0;
      m_monitorElasticPeakPosition = 0;
      m_l1 = 0;
      m_l2 = 0;
      m_supportedInstruments.push_back("IN4");
      m_supportedInstruments.push_back("IN5");
      m_supportedInstruments.push_back("IN6");
    }

    /**
     * Initialise the algorithm
     */
    void LoadILL::init()
    {
      declareProperty(new FileProperty("Filename", "", FileProperty::Load, ".nxs"),
          "File path of the Data file to load");

      declareProperty(new FileProperty("FilenameVanadium", "", FileProperty::OptionalLoad, ".nxs"),
          "File path of the Vanadium file to load (Optional)");

      declareProperty(
          new WorkspaceProperty<API::MatrixWorkspace>("WorkspaceVanadium", "", Direction::Input,
              PropertyMode::Optional), "Vanadium Workspace file to load (Optional)");

      declareProperty(new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
          "The name to use for the output workspace");
```

```
}

/**
 * Execute the algorithm
 */
void LoadILL::exec()
{
  // Retrieve filename
  std::string filenameData = getPropertyValue("Filename");
  std::string filenameVanadium = getPropertyValue("FilenameVanadium");
  MatrixWorkspace_sptr vanaWS = getProperty("WorkspaceVanadium");

  // open the root node
  NeXus::NXRoot dataRoot(filenameData);
  NXEntry dataFirstEntry = dataRoot.openFirstEntry();

  loadInstrumentDetails(dataFirstEntry);
  loadTimeDetails(dataFirstEntry);

  std::vector<std::vector<int> > monitors = getMonitorInfo(dataFirstEntry);

  initWorkSpace(dataFirstEntry, monitors);

  addAllNexusFieldsAsProperties(filenameData);

  runLoadInstrument(); // just to get IDF contents
  initInstrumentSpecific();

  int calculatedDetectorElasticPeakPosition = getEPPFromVanadium(filenameVanadium, vanaWS);

  loadDataIntoTheWorkSpace(dataFirstEntry, monitors, calculatedDetectorElasticPeakPosition);

  addEnergyToRun();
  loadExperimentDetails(dataFirstEntry);

  // load the instrument from the IDF if it exists
  runLoadInstrument();

  // Set the output workspace property
  setProperty("OutputWorkspace", m_localWorkspace);
}

/**
 * Loads Monitor data into an vector of monitor data
 * @return : list of monitor data
 */
std::vector<std::vector<int> > LoadILL::getMonitorInfo(NeXus::NXEntry& firstEntry)
{

  std::vector<std::vector<int> > monitorList;

  for (std::vector<NXClassInfo>::const_iterator it = firstEntry.groups().begin();
      it != firstEntry.groups().end(); ++it)
  {

    if (it->nxclass == "NXmonitor" || boost::starts_with(it->nxname, "monitor"))
    {

      g_log.debug() << "Load monitor data from " + it->nxname;

      NXData dataGroup = firstEntry.openNXData(it->nxname + "/data");
      NXInt data = dataGroup.openIntData();
      // load the counts from the file into memory
      data.load();

      std::vector<int> thisMonitor(data(), data() + data.size());
      monitorList.push_back(thisMonitor);
    }
  }
  return monitorList;
}

/**
 * Get the elastic peak position (EPP) from a Vanadium Workspace
 * or filename.
 * @return the EPP
 */
int LoadILL::getEPPFromVanadium(const std::string &filenameVanadium, MatrixWorkspace_sptr vanaWS)
{
  int calculatedDetectorElasticPeakPosition = -1;

  if (vanaWS != NULL)
  {

    // Check if it has been store on the run object for this workspace
    if (vanaWS->run().hasProperty("EPP"))
    {
      Kernel::Property* prop = vanaWS->run().getProperty("EPP");
      calculatedDetectorElasticPeakPosition = boost::lexical_cast<int>(prop->value());
      g_log.information() << "Using EPP from Vanadium WorkSpace : value =  "
          << calculatedDetectorElasticPeakPosition << "\n";
    }
    else
    {
      g_log.error("No EPP Property in the Vanadium Workspace. Following regular procedure...");
    }
  }
  if (calculatedDetectorElasticPeakPosition == -1 && filenameVanadium != "")
  {
    g_log.information() << "Calculating the elastic peak position from the Vanadium." << std::endl;
    calculatedDetectorElasticPeakPosition = validateVanadium(filenameVanadium);
  }
  return calculatedDetectorElasticPeakPosition;
}

/**
 * Set the instrument name along with its path on the nexus file
 */
void LoadILL::loadInstrumentDetails(NeXus::NXEntry& firstEntry)
{

  m_instrumentPath = m_loader.findInstrumentNexusPath(firstEntry);

  if (m_instrumentPath == "")
```

```
  {
    throw std::runtime_error("Cannot set the instrument name from the Nexus file!");
  }

  m_instrumentName = m_loader.getStringFromNexusPath(firstEntry, m_instrumentPath + "/name");

  if (std::find(m_supportedInstruments.begin(), m_supportedInstruments.end(), m_instrumentName)
      == m_supportedInstruments.end())
  {
    std::string message = "The instrument " + m_instrumentName + " is not valid for this loader!";
    throw std::runtime_error(message);
  }

  g_log.debug() << "Instrument name set to: " + m_instrumentName << std::endl;

}

/**
 * Creates the workspace and initialises member variables with
 * the corresponding values
 *
 * @param entry :: The Nexus entry
 * @param monitors :: list of monitors content
 *
 */
void LoadILL::initWorkSpace(NeXus::NXEntry& entry, const std::vector<std::vector<int> >&monitors)
{

  // read in the data
  NXData dataGroup = entry.openNXData("data");
  NXInt data = dataGroup.openIntData();

  m_numberOfTubes = static_cast<size_t>(data.dim0());
  m_numberOfPixelsPerTube = static_cast<size_t>(data.dim1());
  m_numberOfChannels = static_cast<size_t>(data.dim2());
  size_t numberOfMonitors = monitors.size();

  // dim0 * m_numberOfPixelsPerTube is the total number of detectors
  m_numberOfHistograms = m_numberOfTubes * m_numberOfPixelsPerTube;

  g_log.debug() << "NumberOfTubes: " << m_numberOfTubes << std::endl;
  g_log.debug() << "NumberOfPixelsPerTube: " << m_numberOfPixelsPerTube << std::endl;
  g_log.debug() << "NumberOfChannels: " << m_numberOfChannels << std::endl;

  // Now create the output workspace
  // total number of spectra + number of monitors,
  // bin boundaries = m_numberOfChannels + 1
  // Z/time dimension
  m_localWorkspace = WorkspaceFactory::Instance().create("Workspace2D",
      m_numberOfHistograms + numberOfMonitors, m_numberOfChannels + 1, m_numberOfChannels);
  m_localWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create("TOF");
  m_localWorkspace->setYUnitLabel("Counts");

}

/**
 * Function to do specific instrument stuff
 *
 */
void LoadILL::initInstrumentSpecific()
{
  m_l1 = m_loader.getL1(m_localWorkspace);
  // this will be mainly for IN5 (flat PSD detector)
  m_l2 = m_loader.getInstrumentProperty(m_localWorkspace, "l2");
  if (m_l2 == EMPTY_DBL())
  {
    g_log.debug("Calculating L2 from the IDF.");
    m_l2 = m_loader.getL2(m_localWorkspace);
  }
}

/**
 * Load the time details from the nexus file.
 * @param entry :: The Nexus entry
 */
void LoadILL::loadTimeDetails(NeXus::NXEntry& entry)
{

  m_wavelength = entry.getFloat("wavelength");

  // Monitor can be monitor (IN5) or monitor1 (IN6)
  std::string monitorName;
  if (entry.containsGroup("monitor"))
    monitorName = "monitor";
  else if (entry.containsGroup("monitor1"))
    monitorName = "monitor1";
  else
  {
    std::string message("Cannot find monitor[1] in the Nexus file!");
    g_log.error(message);
    throw std::runtime_error(message);
  }

  m_monitorElasticPeakPosition = entry.getInt(monitorName + "/elasticpeak");

  NXFloat time_of_flight_data = entry.openNXFloat(monitorName + "/time_of_flight");
  time_of_flight_data.load();

  // The entry "monitor/time_of_flight", has 3 fields:
  // channel width , number of channels, Time of flight delay
  m_channelWidth = time_of_flight_data[0];
  //  m_timeOfFlightDelay = time_of_flight_data[2];

  g_log.debug("Nexus Data:");
  g_log.debug() << " ChannelWidth: " << m_channelWidth << std::endl;
  g_log.debug() << " Wavelength: " << m_wavelength << std::endl;
  g_log.debug() << " ElasticPeakPosition: " << m_monitorElasticPeakPosition << std::endl;

}

/**
 * Goes through all the fields of the nexus file and add them
 * as parameters in the workspace
 * @param filename :: NeXus file
 */
```

```
void LoadILL::addAllNexusFieldsAsProperties(std::string filename)
{

  API::Run & runDetails = m_localWorkspace->mutableRun();

  // Open NeXus file
  NXhandle nxfileID;
  NXstatus stat = NXopen(filename.c_str(), NXACC_READ, &nxfileID);

  g_log.debug() << "Starting parsing properties from : " << filename << std::endl;
  if (stat == NX_ERROR)
  {
    g_log.debug() << "convertNexusToProperties: Error loading " << filename;
    throw Kernel::Exception::FileError("Unable to open File:", filename);
  }
  m_loader.addNexusFieldsToWsRun(nxfileID, runDetails);

  g_log.debug() << "End parsing properties from : " << filename << std::endl;

  // Add also "Facility", as asked
  runDetails.addProperty("Facility", std::string("ILL"));

  stat = NXclose(&nxfileID);

}

/**
 * Calculates the Energy from the wavelength and adds
 * it at property Ei
 */
void LoadILL::addEnergyToRun()
{

  API::Run & runDetails = m_localWorkspace->mutableRun();
  double ei = m_loader.calculateEnergy(m_wavelength);
  runDetails.addProperty<double>("Ei", ei, true); //overwrite

}

/*
 * Load data about the Experiment.
 *
 * TODO: This is very incomplete. We need input from scientists to complete the code below
 *
 * @param entry :: The Nexus entry
 */
void LoadILL::loadExperimentDetails(NXEntry & entry)
{

  // TODO: Do the rest
  // Pick out the geometry information

  std::string description = boost::lexical_cast<std::string>(entry.getFloat("sample/description"));

  m_localWorkspace->mutableSample().setName(description);

  //  m_localWorkspace->mutableSample().setThickness(static_cast<double> (isis_raw->spb.e_thick));
  //  m_localWorkspace->mutableSample().setHeight(static_cast<double> (isis_raw->spb.e_height));
  //  m_localWorkspace->mutableSample().setWidth(static_cast<double> (isis_raw->spb.e_width));

}

/**
 * Gets the experimental Elastic Peak Position in the dectector
 * as the value parsed from the nexus file might be wrong.
 *
 * It gets a few spectra in the equatorial line of the detector,
 * sum them up and finds the maximum = the Elastic peak
 *
 *
 * @param data :: spectra data
 * @return detector Elastic Peak Position
 */
int LoadILL::getDetectorElasticPeakPosition(const NeXus::NXInt &data)
{

  // j = index in the equatorial line (256/2=128)
  // both index 127 and 128 are in the equatorial line
  size_t j = m_numberOfPixelsPerTube / 2;

  // ignore the first tubes and the last ones to avoid the beamstop
  //get limits in the m_numberOfTubes
  size_t tubesToRemove = m_numberOfTubes / 7;

  std::vector<int> cumulatedSumOfSpectras(m_numberOfChannels, 0);
  for (size_t i = tubesToRemove; i < m_numberOfTubes - tubesToRemove; i++)
  {
    int* data_p = &data(static_cast<int>(i), static_cast<int>(j), 0);
    std::vector<int> thisSpectrum(data_p, data_p + m_numberOfChannels);
    // sum spectras
    std::transform(thisSpectrum.begin(), thisSpectrum.end(), cumulatedSumOfSpectras.begin(),
        cumulatedSumOfSpectras.begin(), std::plus<int>());
  }
  auto it = std::max_element(cumulatedSumOfSpectras.begin(), cumulatedSumOfSpectras.end());

  int calculatedDetectorElasticPeakPosition;
  if (it == cumulatedSumOfSpectras.end())
  {
    g_log.warning() << "No Elastic peak position found! Assuming the EPP in the Nexus file: "
        << m_monitorElasticPeakPosition << std::endl;
    calculatedDetectorElasticPeakPosition = m_monitorElasticPeakPosition;

  }
  else
  {
    //calculatedDetectorElasticPeakPosition = *it;
    calculatedDetectorElasticPeakPosition = static_cast<int>(std::distance(
        cumulatedSumOfSpectras.begin(), it));

    if (calculatedDetectorElasticPeakPosition == 0)
    {
      g_log.warning() << "Elastic peak position is ZERO Assuming the EPP in the Nexus file: "
          << m_monitorElasticPeakPosition << std::endl;
      calculatedDetectorElasticPeakPosition = m_monitorElasticPeakPosition;
```

```
    }
    else
    {
      g_log.debug() << "Calculated Detector EPP: " << calculatedDetectorElasticPeakPosition;
      g_log.debug() << " :: Read EPP from the nexus file: " << m_monitorElasticPeakPosition
          << std::endl;
    }
  }
  return calculatedDetectorElasticPeakPosition;

}

/**
 * It loads the vanadium nexus file and cross checks it against the
 * data file already loaded (same wavelength and same instrument configuration).
 * If matches looks for the elastic peak in the vanadium file and returns
 * it position.
 *
 * @param filenameVanadium :: The path for the vanadium nexus file.
 * @return The elastic peak position inside the tof channels.
 */
int LoadILL::validateVanadium(const std::string &filenameVanadium)
{
  NeXus::NXRoot vanaRoot(filenameVanadium);
  NXEntry vanaFirstEntry = vanaRoot.openFirstEntry();

  double wavelength = vanaFirstEntry.getFloat("wavelength");

  // read in the data
  NXData dataGroup = vanaFirstEntry.openNXData("data");
  NXInt data = dataGroup.openIntData();

  size_t numberOfTubes = static_cast<size_t>(data.dim0());
  size_t numberOfPixelsPerTube = static_cast<size_t>(data.dim1());
  size_t numberOfChannels = static_cast<size_t>(data.dim2());

  if (wavelength != m_wavelength || numberOfTubes != m_numberOfTubes
      || numberOfPixelsPerTube != m_numberOfPixelsPerTube || numberOfChannels != m_numberOfChannels)
  {
    throw std::runtime_error("Vanadium and Data were not collected in the same conditions!");
  }

  data.load();
  int calculatedDetectorElasticPeakPosition = getDetectorElasticPeakPosition(data);
  return calculatedDetectorElasticPeakPosition;
}

/**
 * Loads all the spectra into the workspace, including that from the monitor
 *
 * @param entry :: The Nexus entry
 * @param monitors :: List of monitors content
 * @param vanaCalculatedDetectorElasticPeakPosition :: If -1 uses this value as the elastic peak position at the detector.
 *
 */
void LoadILL::loadDataIntoTheWorkSpace(NeXus::NXEntry& entry,
    const std::vector<std::vector<int> >&monitors, int vanaCalculatedDetectorElasticPeakPosition)
{

  // read in the data
  NXData dataGroup = entry.openNXData("data");
  NXInt data = dataGroup.openIntData();
  // load the counts from the file into memory
  data.load();

  // Detector: Find real elastic peak in the detector.
  // Looks for a few elastic peaks on the equatorial line of the detector.
  int calculatedDetectorElasticPeakPosition;
  if (vanaCalculatedDetectorElasticPeakPosition == -1)
    calculatedDetectorElasticPeakPosition = getDetectorElasticPeakPosition(data);
  else
    calculatedDetectorElasticPeakPosition = vanaCalculatedDetectorElasticPeakPosition;

  //set it as a Property
  API::Run & runDetails = m_localWorkspace->mutableRun();
  runDetails.addProperty("EPP", calculatedDetectorElasticPeakPosition);

  double theoreticalElasticTOF = (m_loader.calculateTOF(m_l1, m_wavelength)
      + m_loader.calculateTOF(m_l2, m_wavelength)) * 1e6; //microsecs

  // Calculate the real tof (t1+t2) put it in tof array
  std::vector<double> detectorTofBins(m_numberOfChannels + 1);
  for (size_t i = 0; i < m_numberOfChannels + 1; ++i)
  {
    detectorTofBins[i] = theoreticalElasticTOF
        + m_channelWidth
            * static_cast<double>(static_cast<int>(i) - calculatedDetectorElasticPeakPosition)
        - m_channelWidth / 2; // to make sure the bin is in the middle of the elastic peak
  }

  g_log.information() << "T1+T2 : Theoretical = " << theoreticalElasticTOF;
  g_log.information() << " ::  Calculated bin = ["
      << detectorTofBins[calculatedDetectorElasticPeakPosition] << ","
      << detectorTofBins[calculatedDetectorElasticPeakPosition + 1] << "]" << std::endl;

  // The binning for monitors is considered the same as for detectors
  size_t spec = 0;

  for (auto it = monitors.begin(); it != monitors.end(); ++it)
  {

    m_localWorkspace->dataX(spec).assign(detectorTofBins.begin(), detectorTofBins.end());
    // Assign Y
    m_localWorkspace->dataY(spec).assign(it->begin(), it->end());
    // Assign Error
    MantidVec& E = m_localWorkspace->dataE(spec);
    std::transform(it->begin(), it->end(), E.begin(), LoadILL::calculateError);
    ++spec;

  }

  // Assign calculated bins to first X axis
  size_t firstSpec = spec;
  m_localWorkspace->dataX(firstSpec).assign(detectorTofBins.begin(), detectorTofBins.end());
```

```
    Progress progress(this, 0, 1, m_numberOfTubes * m_numberOfPixelsPerTube);
    //size_t spec = 0;
    for (size_t i = 0; i < m_numberOfTubes; ++i)
    {
      for (size_t j = 0; j < m_numberOfPixelsPerTube; ++j)
      {
        if (spec > firstSpec)
        {
          // just copy the time binning axis to every spectra
          m_localWorkspace->dataX(spec) = m_localWorkspace->readX(firstSpec);
        }
        // Assign Y
        int* data_p = &data(static_cast<int>(i), static_cast<int>(j), 0);
        m_localWorkspace->dataY(spec).assign(data_p, data_p + m_numberOfChannels);

        // Assign Error
        MantidVec& E = m_localWorkspace->dataE(spec);
        std::transform(data_p, data_p + m_numberOfChannels, E.begin(), LoadILL::calculateError);

        ++spec;
        progress.report();
      }
    }
  }

  /**
   * Run the Child Algorithm LoadInstrument.
   */
  void LoadILL::runLoadInstrument()
  {

    IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");

    // Now execute the Child Algorithm. Catch and log any error, but don't stop.
    try
    {

      // TODO: depending on the m_numberOfPixelsPerTube we might need to load a different IDF

      loadInst->setPropertyValue("InstrumentName", m_instrumentName);
      loadInst->setProperty<MatrixWorkspace_sptr>("Workspace", m_localWorkspace);
      loadInst->execute();
    } catch (...)
    {
      g_log.information("Cannot load the instrument definition.");
    }
  }

} // namespace DataHandling
} // namespace Mantid
```

LoadILLAscii	Loads ILL Raw data in Ascii format.	ILL: D2B

```
/*WIKI*

 Loads an ILL Ascii / Raw data file into a [[Workspace2D]] with the given name.
 To date this Loader is only compatible with non TOF instruments.

 Supported instruments : ILL D2B

 *WIKI*/

#include "MantidMDAlgorithms/LoadILLAscii.h"
#include "MantidAPI/FileProperty.h"
#include "MantidGeometry/Instrument/ComponentHelper.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidMDAlgorithms/LoadILLAsciiHelper.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidKernel/System.h"
#include "MantidKernel/DateAndTime.h"
#include "MantidKernel/System.h"
#include "MantidAPI/FileProperty.h"
#include "MantidGeometry/MDGeometry/MDHistoDimension.h"
#include "MantidAPI/IMDEventWorkspace.h"
#include "MantidKernel/TimeSeriesProperty.h"

#include <algorithm>
#include <iterator>     // std::distance
#include <sstream>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <string.h>

#include <boost/shared_ptr.hpp>
#include <Poco/TemporaryFile.h>

namespace Mantid {
namespace MDAlgorithms {

using namespace Kernel;
using namespace API;

// Register the algorithm into the AlgorithmFactory
DECLARE_FILELOADER_ALGORITHM(LoadILLAscii)

//----------------------------------------------------------------------------------------------
/** Constructor
 */
LoadILLAscii::LoadILLAscii() :
                m_instrumentName(""), m_wavelength(0) {
        // Add here supported instruments by this loader
        m_supportedInstruments.push_back("D2B");
}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
LoadILLAscii::~LoadILLAscii() {
}

/**
 * Return the confidence with with this algorithm can load the file
 * @param descriptor A descriptor for the file
 * @returns An integer specifying the confidence level. 0 indicates it will not be used
 */
int LoadILLAscii::confidence(Kernel::FileDescriptor & descriptor) const {
        const std::string & filePath = descriptor.filename();
        // Avoid some known file types that have different loaders
        int confidence(0);

        if (descriptor.isAscii()) {
                confidence = 10; // Low so that others may try
                ILLParser p(filePath);
                std::string instrumentName = p.getInstrumentName();

                g_log.information() << "Instrument name: " << instrumentName << "\n";

                if (std::find(m_supportedInstruments.begin(),
                                m_supportedInstruments.end(), instrumentName)
                                != m_supportedInstruments.end())
                        confidence = 80;
        }

        return confidence;
}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string LoadILLAscii::name() const {
        return "LoadILLAscii";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int LoadILLAscii::version() const {
        return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string LoadILLAscii::category() const {
        return "MDAlgorithms\\Text";
}

//----------------------------------------------------------------------------------------------
/// Summary of behaviour
```

```
const std::string LoadILLAscii::summary() const
{
	return "Loads ILL Raw data in Ascii format.";
}

//----------------------------------------------------------------------------------------------
/** Initialize the algorithm's properties.
 */
void LoadILLAscii::init() {
	declareProperty(new FileProperty("Filename", "", FileProperty::Load, ""),
			"Name of the data file to load.");
	declareProperty(
			new WorkspaceProperty<IMDEventWorkspace>("OutputWorkspace", "",
					Direction::Output), "Name to use for the output workspace.");

}

//----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void LoadILLAscii::exec() {

	std::string filename = getPropertyValue("Filename");

	// Parses ascii file and fills the data scructures
	ILLParser illAsciiParser(filename);
	loadInstrumentName(illAsciiParser);
	illAsciiParser.parse();
	loadExperimentDetails(illAsciiParser);

	// get local references to the parsed file
	const std::vector<std::vector<int> > &spectraList = illAsciiParser.getSpectraList();
	const std::vector<std::map<std::string, std::string> > &spectraHeaderList = illAsciiParser.getSpectraHeaderList();

	// list containing all parsed scans. 1 scan => 1 ws
	std::vector<API::MatrixWorkspace_sptr> workspaceList;
	workspaceList.reserve(spectraList.size());

	// iterate parsed file
	std::vector<std::vector<int> >::const_iterator iSpectra;
	std::vector<std::map<std::string, std::string> >::const_iterator iSpectraHeader;

	Progress progress(this, 0, 1, spectraList.size());
	for (iSpectra = spectraList.begin(), iSpectraHeader = spectraHeaderList.begin();
			iSpectra < spectraList.end() && iSpectraHeader < spectraHeaderList.end();
			++iSpectra, ++iSpectraHeader) {

		size_t spectrumIndex = std::distance(spectraList.begin(), iSpectra);
		g_log.debug() << "Reading Spectrum: " << spectrumIndex << std::endl;

		std::vector<int> thisSpectrum = *iSpectra;
		API::MatrixWorkspace_sptr thisWorkspace = WorkspaceFactory::Instance().create("Workspace2D",
				thisSpectrum.size(), 2, 1);

		thisWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create("Wavelength");
		thisWorkspace->setYUnitLabel("Counts");

		// Set this workspace position
		double currentPositionAngle = illAsciiParser.getValue<double>("angles*1000", *iSpectraHeader) / 1000;
		setWorkspaceRotationAngle(thisWorkspace,currentPositionAngle);

		//
		loadsDataIntoTheWS(thisWorkspace, thisSpectrum);
		loadIDF(thisWorkspace); // assigns data to the instrument

		workspaceList.push_back(thisWorkspace);

//		// just to see the list of WS in MantidPlot if needed for debugging
//		std::stringstream outWorkspaceNameStream;
//		outWorkspaceNameStream << "test" << std::distance(spectraList.begin(), iSpectra);
//		AnalysisDataService::Instance().addOrReplace(outWorkspaceNameStream.str(), thisWorkspace);


		progress.report("Loading scans...");
	}

	// Merge the workspace list into a single WS with a virtual instrument
	IMDEventWorkspace_sptr outWorkspace = mergeWorkspaces(workspaceList);
	setProperty("OutputWorkspace",outWorkspace);


}

/**
 * Sets the workspace position based on the rotation angle
 * See tag logfile in file instrument/D2B_Definition.xml
 */
void LoadILLAscii::setWorkspaceRotationAngle(API::MatrixWorkspace_sptr ws, double rotationAngle){

	API::Run & runDetails = ws->mutableRun();
	auto *p = new Mantid::Kernel::TimeSeriesProperty<double>("rotangle");

//	auto p = boost::make_shared <Mantid::Kernel::TimeSeriesProperty<double> >("rotangle");

	p->addValue(DateAndTime::getCurrentTime(), rotationAngle);
	runDetails.addLogData(p);
}

/**
 * Loads instrument details
 */
void LoadILLAscii::loadExperimentDetails(ILLParser &p) {

	m_wavelength = p.getValueFromHeader<double>("wavelength");
	g_log.debug() << "Wavelength: " << m_wavelength << std::endl;
}

void LoadILLAscii::loadInstrumentName(ILLParser &p) {

	m_instrumentName = p.getInstrumentName();
	if (m_instrumentName == "") {
		throw std::runtime_error(
				"Cannot read instrument name from the data file.");
	}
	g_log.debug() << "Instrument name set to: " + m_instrumentName << std::endl;
```

```
        m_wavelength = p.getValueFromHeader<double>("wavelength");
        g_log.debug() << "Wavelength: " << m_wavelength << std::endl;
}

/**
 * Run the Child Algorithm LoadInstrument.
 */
void LoadILLAscii::loadIDF(API::MatrixWorkspace_sptr &workspace) {

        IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");
        // Now execute the Child Algorithm. Catch and log any error, but don't stop.
        try {
                loadInst->setPropertyValue("InstrumentName", m_instrumentName);
                loadInst->setProperty<MatrixWorkspace_sptr>("Workspace", workspace);
                loadInst->execute();
        } catch (...) {
                g_log.information("Cannot load the instrument definition.");
        }
}

/**
 * Loads the scan into the workspace
 */
void LoadILLAscii::loadsDataIntoTheWS(API::MatrixWorkspace_sptr &thisWorkspace,
                const std::vector<int> &thisSpectrum) {

        thisWorkspace->dataX(0)[0] = m_wavelength - 0.001;
        thisWorkspace->dataX(0)[1] = m_wavelength + 0.001;

        size_t spec = 0;
        for (size_t i = 0; i < thisSpectrum.size(); ++i) {

                if (spec > 0) {
                        // just copy the time binning axis to every spectra
                        thisWorkspace->dataX(spec) = thisWorkspace->readX(0);
                }
                // Assign Y
                thisWorkspace->dataY(spec)[0] = thisSpectrum[i];
                // Assign Error
                thisWorkspace->dataE(spec)[0] = thisSpectrum[i] * thisSpectrum[i];

                ++spec;
        }

        loadIDF(thisWorkspace); // assigns data to the instrument

}


/**
 * Merge all workspaces and create a virtual new instrument.
 *
 * To date this is slow as we are passing through a temp file and then
 * it is loaded in the ImportMDEventWorkspace.
 * If this loader is to used at the ILL, the better option is to avoid
 * a MDWS and go ahead with the merge instruments.
 *
 * @return MD Event workspace
 *
 */
IMDEventWorkspace_sptr LoadILLAscii::mergeWorkspaces(
                std::vector<API::MatrixWorkspace_sptr> &workspaceList) {

        Poco::TemporaryFile tmpFile;
        std::string tempFileName = tmpFile.path();
        g_log.debug() << "Dumping WSs in a temp file: " << tempFileName << std::endl;


        std::ofstream myfile;
        myfile.open (tempFileName.c_str());
        myfile << "DIMENSIONS" <<std::endl;
        myfile << "x X m 100" <<std::endl;
        myfile << "y Y m 100" <<std::endl;
        myfile << "z Z m 100" <<std::endl;
        myfile << "# Signal, Error, DetectorId, RunId, coord1, coord2, ... to end of coords" <<std::endl;
        myfile << "MDEVENTS" <<std::endl;


        if (workspaceList.size() > 0) {
                Progress progress(this, 0, 1, workspaceList.size());
                for (auto it = workspaceList.begin(); it < workspaceList.end(); ++it) {
                        std::size_t pos = std::distance(workspaceList.begin(),it);
                        API::MatrixWorkspace_sptr thisWorkspace = *it;

                        std::size_t nHist = thisWorkspace->getNumberHistograms();
                        for (std::size_t i=0; i < nHist; ++i){
                                Geometry::IDetector_const_sptr det = thisWorkspace->getDetector(i);
                                const MantidVec& signal = thisWorkspace->readY(i);
                                const MantidVec& error = thisWorkspace->readE(i);
                                myfile << signal[0] << " ";
                                myfile << error[0] << " ";
                                myfile << det->getID() << " ";
                                myfile << pos << " ";
                                Kernel::V3D detPos = det->getPos();
                                myfile << detPos.X() << " ";
                                myfile << detPos.Y() << " ";
                                myfile << detPos.Z() << " ";
                                myfile << std::endl;
                        }
                        progress.report("Creating MD WS");
                }
                myfile.close();


                IAlgorithm_sptr importMDEWS = createChildAlgorithm("ImportMDEventWorkspace");
                // Now execute the Child Algorithm.
                try {
                        importMDEWS->setPropertyValue("Filename", tempFileName);
                        importMDEWS->setProperty("OutputWorkspace", "Test");
                        importMDEWS->executeAsChildAlg();
                } catch (std::exception & exc) {
                        throw std::runtime_error(std::string("Error running ImportMDEventWorkspace: ") + exc.what());
                }
                IMDEventWorkspace_sptr workspace = importMDEWS->getProperty("OutputWorkspace");
```

```
                if(!workspace)
                        throw(std::runtime_error("Can not retrieve results of child algorithm ImportMDEventWorkspace"));

                return workspace;

        }
        else{
                throw std::runtime_error("Error: No workspaces were found to be merged!");
        }

}

} // namespace MDAlgorithms
} // namespace Mantid
```

LoadILLIndirect	Loads a ILL/IN16B nexus file.	ILL: IN16B

```
#include "MantidDataHandling/LoadILLIndirect.h"
#include "MantidAPI/FileProperty.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidGeometry/Instrument/ComponentHelper.h"

#include <boost/algorithm/string.hpp>

#include <nexus/napi.h>
#include <iostream>
#include <iomanip>      // std::setw

namespace Mantid {
namespace DataHandling {

using namespace Kernel;
using namespace API;
using namespace NeXus;

// Register the algorithm into the AlgorithmFactory
DECLARE_NEXUS_FILELOADER_ALGORITHM (LoadILLIndirect);



//----------------------------------------------------------------------------------------------
/** Constructor
 */
LoadILLIndirect::LoadILLIndirect() :
                API::IFileLoader<Kernel::NexusDescriptor>(),
                m_numberOfTubes(0),
                m_numberOfPixelsPerTube(0),
                m_numberOfChannels(0),
                m_numberOfSimpleDetectors(0),
                m_numberOfHistograms(0){
        m_supportedInstruments.push_back("IN16B");
}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
LoadILLIndirect::~LoadILLIndirect() {
}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string LoadILLIndirect::name() const {
        return "LoadILLIndirect";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int LoadILLIndirect::version() const {
        return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string LoadILLIndirect::category() const {
        return "DataHandling";
}

 //----------------------------------------------------------------------------------------------


/**
* Return the confidence with with this algorithm can load the file
* @param descriptor A descriptor for the file
* @returns An integer specifying the confidence level. 0 indicates it will not be used
*/
int LoadILLIndirect::confidence(Kernel::NexusDescriptor & descriptor) const
{

  // fields existent only at the ILL
  if (descriptor.pathExists("/entry0/wavelength")// ILL
    && descriptor.pathExists("/entry0/experiment_identifier")// ILL
    && descriptor.pathExists("/entry0/mode")// ILL
    && descriptor.pathExists("/entry0/dataSD/dataSD")// IN16B
    && descriptor.pathExists("/entry0/instrument/Doppler/doppler_frequency")// IN16B
  ) {
      return 80;
  }
  else
  {
    return 0;
  }
}

 //----------------------------------------------------------------------------------------------
/** Initialize the algorithm's properties.
 */
void LoadILLIndirect::init() {
    declareProperty(
      new FileProperty("Filename", "", FileProperty::Load, ".nxs"),
      "File path of the Data file to load");

    declareProperty(
      new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
      "The name to use for the output workspace");
}



 //----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void LoadILLIndirect::exec() {
    // Retrieve filename
```

```
    std::string filenameData = getPropertyValue("Filename");

    // open the root node
    NeXus::NXRoot dataRoot(filenameData);
    NXEntry firstEntry = dataRoot.openFirstEntry();

    // Load Monitor details: n. monitors x monitor contents
    std::vector< std::vector<int> > monitorsData = loadMonitors(firstEntry);

    // Load Data details (number of tubes, channels, etc)
    loadDataDetails(firstEntry);

        std::string instrumentPath = m_loader.findInstrumentNexusPath(firstEntry);
        setInstrumentName(firstEntry, instrumentPath);

    initWorkSpace(firstEntry, monitorsData);

        g_log.debug("Building properties...");
    loadNexusEntriesIntoProperties(filenameData);

        g_log.debug("Loading data...");
    loadDataIntoTheWorkSpace(firstEntry, monitorsData);

    // load the instrument from the IDF if it exists
        g_log.debug("Loading instrument definition...");
    runLoadInstrument();

    //moveSingleDetectors(); Work in progress

    // Set the output workspace property
    setProperty("OutputWorkspace", m_localWorkspace);
}

/**
* Set member variable with the instrument name
*/
void LoadILLIndirect::setInstrumentName(const NeXus::NXEntry &firstEntry,
                const std::string &instrumentNamePath) {

        if (instrumentNamePath == "") {
                std::string message(
                                "Cannot set the instrument name from the Nexus file!");
                g_log.error(message);
                throw std::runtime_error(message);
        }
        m_instrumentName = m_loader.getStringFromNexusPath(firstEntry,
                        instrumentNamePath + "/name");
        boost::to_upper(m_instrumentName);// "IN16b" in file, keep it upper case.
        g_log.debug() << "Instrument name set to: " + m_instrumentName << std::endl;

}

/**
* Load Data details (number of tubes, channels, etc)
* @param entry First entry of nexus file
*/
void LoadILLIndirect::loadDataDetails(NeXus::NXEntry& entry)
{
        // read in the data
        NXData dataGroup = entry.openNXData("data");
        NXInt data = dataGroup.openIntData();

        m_numberOfTubes = static_cast<size_t>(data.dim0());
        m_numberOfPixelsPerTube = static_cast<size_t>(data.dim1());
        m_numberOfChannels = static_cast<size_t>(data.dim2());


        NXData dataSDGroup = entry.openNXData("dataSD");
        NXInt dataSD = dataSDGroup.openIntData();

        m_numberOfSimpleDetectors = static_cast<size_t>(dataSD.dim0());
}


/**
   * Load monitors data found in nexus file
   *
   * @param entry :: The Nexus entry
   *
   */
std::vector< std::vector<int> > LoadILLIndirect::loadMonitors(NeXus::NXEntry& entry){
        // read in the data
        g_log.debug("Fetching monitor data...");

        NXData dataGroup = entry.openNXData("monitor/data");
        NXInt data = dataGroup.openIntData();
        // load the counts from the file into memory
        data.load();

        // For the moment, we are aware of only one monitor entry, but we keep the generalized case of n monitors

        std::vector< std::vector<int> > monitors(1);
        std::vector<int> monitor(data(), data()+data.size());
        monitors[0].swap(monitor);
        return monitors;
}




/**
   * Creates the workspace and initialises member variables with
   * the corresponding values
   *
   * @param entry :: The Nexus entry
   * @param monitorsData :: Monitors data already loaded
   *
   */
void LoadILLIndirect::initWorkSpace(NeXus::NXEntry& /*entry*/, std::vector< std::vector<int> > monitorsData)
{

        // dim0 * m_numberOfPixelsPerTube is the total number of detectors
        m_numberOfHistograms = m_numberOfTubes * m_numberOfPixelsPerTube;

        g_log.debug() << "NumberOfTubes: " << m_numberOfTubes << std::endl;
```

```
        g_log.debug() << "NumberOfPixelsPerTube: " << m_numberOfPixelsPerTube << std::endl;
        g_log.debug() << "NumberOfChannels: " << m_numberOfChannels << std::endl;
        g_log.debug() << "NumberOfSimpleDetectors: " << m_numberOfSimpleDetectors << std::endl;
        g_log.debug() << "Monitors: " << monitorsData.size() << std::endl;
        g_log.debug() << "Monitors[0]: " << monitorsData[0].size() << std::endl;

        // Now create the output workspace

        m_localWorkspace = WorkspaceFactory::Instance().create(
                        "Workspace2D",
                        m_numberOfHistograms+monitorsData.size()+m_numberOfSimpleDetectors,
                        m_numberOfChannels + 1,
                        m_numberOfChannels);

        m_localWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create("Empty");

        m_localWorkspace->setYUnitLabel("Counts");

}


/**
   * Load data found in nexus file
   *
   * @param entry :: The Nexus entry
   * @param monitorsData :: Monitors data already loaded
   *
   */
void LoadILLIndirect::loadDataIntoTheWorkSpace(NeXus::NXEntry& entry, std::vector< std::vector<int> > monitorsData)
{

  // read in the data
  NXData dataGroup = entry.openNXData("data");
  NXInt data = dataGroup.openIntData();
  // load the counts from the file into memory
  data.load();

  // Same for Simple Detectors
  NXData dataSDGroup = entry.openNXData("dataSD");
  NXInt dataSD = dataSDGroup.openIntData();
  // load the counts from the file into memory
  dataSD.load();


  // Assign calculated bins to first X axis
////  m_localWorkspace->dataX(0).assign(detectorTofBins.begin(), detectorTofBins.end());

  size_t spec = 0;
  size_t nb_monitors = monitorsData.size();
  size_t nb_SD_detectors = dataSD.dim0();

  Progress progress(this, 0, 1, m_numberOfTubes * m_numberOfPixelsPerTube + nb_monitors + nb_SD_detectors);

  // Assign fake values to first X axis <<to be completed>>
  for (size_t i = 0; i <= m_numberOfChannels; ++i) {
          m_localWorkspace->dataX(0)[i] = double(i);
  }

  // First, Monitor
  for (size_t im = 0; im<nb_monitors; im++){

      if (im > 0)
      {
        m_localWorkspace->dataX(im) = m_localWorkspace->readX(0);
      }

      // Assign Y
      int* monitor_p = monitorsData[im].data();
      m_localWorkspace->dataY(im).assign(monitor_p, monitor_p + m_numberOfChannels);

          progress.report();
  }


  // Then Tubes
  for (size_t i = 0; i < m_numberOfTubes; ++i)
  {
    for (size_t j = 0; j < m_numberOfPixelsPerTube; ++j)
    {

      // just copy the time binning axis to every spectra
      m_localWorkspace->dataX(spec+nb_monitors) = m_localWorkspace->readX(0);

      // Assign Y
      int* data_p = &data(static_cast<int>(i), static_cast<int>(j), 0);
      m_localWorkspace->dataY(spec+nb_monitors).assign(data_p, data_p + m_numberOfChannels);

      // Assign Error
      MantidVec& E = m_localWorkspace->dataE(spec+nb_monitors);
      std::transform(data_p, data_p + m_numberOfChannels, E.begin(),
                LoadILLIndirect::calculateError);

      ++spec;
      progress.report();
    }
  }// for m_numberOfTubes

  // Then add Simple Detector (SD)
  for (int i = 0; i < dataSD.dim0(); ++i) {

      // just copy again the time binning axis to every spectra
      m_localWorkspace->dataX(spec+nb_monitors+i) = m_localWorkspace->readX(0);

      // Assign Y
      int* dataSD_p = &dataSD(i, 0, 0);
      m_localWorkspace->dataY(spec+nb_monitors+i).assign(dataSD_p, dataSD_p + m_numberOfChannels);

          progress.report();
  }

}// LoadILLIndirect::loadDataIntoTheWorkSpace
```

```
void LoadILLIndirect::loadNexusEntriesIntoProperties(std::string nexusfilename) {

    API::Run & runDetails = m_localWorkspace->mutableRun();

    // Open NeXus file
    NXhandle nxfileID;
    NXstatus stat=NXopen(nexusfilename.c_str(), NXACC_READ, &nxfileID);

    if(stat==NX_ERROR)
    {
        g_log.debug() << "convertNexusToProperties: Error loading " << nexusfilename;
        throw Kernel::Exception::FileError("Unable to open File:" , nexusfilename);
    }
    m_loader.addNexusFieldsToWsRun(nxfileID, runDetails);

    // Add also "Facility", as asked
    runDetails.addProperty("Facility", std::string("ILL"));

    stat=NXclose(&nxfileID);
}


/**
   * Run the Child Algorithm LoadInstrument.
   */
void LoadILLIndirect::runLoadInstrument() {

        IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");

        // Now execute the Child Algorithm. Catch and log any error, but don't stop.
        try {
                loadInst->setPropertyValue("InstrumentName", m_instrumentName);
                loadInst->setProperty<MatrixWorkspace_sptr>("Workspace", m_localWorkspace);
                loadInst->execute();

        } catch (...) {
                g_log.information("Cannot load the instrument definition.");
        }
}

void LoadILLIndirect::moveComponent(const std::string &componentName, double twoTheta, double offSet) {

        try {

                Geometry::Instrument_const_sptr instrument = m_localWorkspace->getInstrument();
                Geometry::IComponent_const_sptr component = instrument->getComponentByName(componentName);

                double r, theta, phi, newTheta, newR;
                V3D oldPos = component->getPos();
                oldPos.getSpherical(r, theta, phi);

                newTheta = twoTheta;
                newR = offSet;

                V3D newPos;
                newPos.spherical(newR, newTheta, phi);

                //g_log.debug() << tube->getName() << " : t = " << theta << " ==> t = " << newTheta << "\n";
                Geometry::ParameterMap& pmap = m_localWorkspace->instrumentParameters();
                Geometry::ComponentHelper::moveComponent(*component, pmap, newPos, Geometry::ComponentHelper::Absolute);


        } catch (Mantid::Kernel::Exception::NotFoundError&) {
                throw std::runtime_error(
                                "Error when trying to move the "  + componentName +  " : NotFoundError");
        } catch (std::runtime_error &) {
                throw std::runtime_error(
                                "Error when trying to move the "  + componentName +  " : runtime_error");
        }

}

/**
 * IN16B has a few single detectors that are place around the sample.
 * They are moved according to some values in the nexus file.
 * This is not implemented yet.
 */
void LoadILLIndirect::moveSingleDetectors(){


        std::string prefix("single_tube_");

        for (int i=1; i<=8; i++){
                std::string componentName = prefix + boost::lexical_cast<std::string>(i);

                moveComponent(componentName, i*20.0, 2.0+i/10.0);

        }


}


} // namespace DataHandling
} // namespace Mantid
```

LoadILLReflectometry	Loads a ILL/D17 nexus file.	ILL: D17

```
/*WIKI*
 TODO: Enter a full wiki-markup description of your algorithm here. You can then use the Build/wiki_maker.py script to generate your full
wiki page.
 *WIKI*/

#include "MantidDataHandling/LoadILLReflectometry.h"
#include "MantidAPI/FileProperty.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidGeometry/Instrument/ComponentHelper.h"

#include <boost/algorithm/string.hpp>

#include <nexus/napi.h>
#include <iostream>

namespace Mantid
{
  namespace DataHandling
  {

    using namespace Kernel;
    using namespace API;
    using namespace NeXus;

// Register the algorithm into the AlgorithmFactory
    DECLARE_NEXUS_FILELOADER_ALGORITHM(LoadILLReflectometry);

// PI again !
    const double PI = 3.14159265358979323846264338327950288419716939937510582;

//----------------------------------------------------------------------------------------------
    /** Constructor
     */
    LoadILLReflectometry::LoadILLReflectometry()
    {
      m_numberOfTubes = 0; // number of tubes - X
      m_numberOfPixelsPerTube = 0; //number of pixels per tube - Y
      m_numberOfChannels = 0; // time channels - Z
      m_numberOfHistograms = 0;
      m_supportedInstruments.push_back("D17");
    }

//----------------------------------------------------------------------------------------------
    /** Destructor
     */
    LoadILLReflectometry::~LoadILLReflectometry()
    {
    }

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
    const std::string LoadILLReflectometry::name() const
    {
      return "LoadILLReflectometry";
    }
    ;

/// Algorithm's version for identification. @see Algorithm::version
    int LoadILLReflectometry::version() const
    {
      return 1;
    }
    ;

/// Algorithm's category for identification. @see Algorithm::category
    const std::string LoadILLReflectometry::category() const
    {
      return "DataHandling";
    }

    /**
     * Return the confidence with with this algorithm can load the file
     * @param descriptor A descriptor for the file
     * @returns An integer specifying the confidence level. 0 indicates it will not be used
     */
    int LoadILLReflectometry::confidence(Kernel::NexusDescriptor & descriptor) const
    {

      // fields existent only at the ILL
      if (descriptor.pathExists("/entry0/wavelength")  // ILL
      && descriptor.pathExists("/entry0/experiment_identifier")  // ILL
          && descriptor.pathExists("/entry0/mode")  // ILL
          && descriptor.pathExists("/entry0/instrument/Chopper1") // TO BE DONE
          && descriptor.pathExists("/entry0/instrument/Chopper2")  // ???
              )
      {
        return 80;
      }
      else
      {
        return 0;
      }
    }

//----------------------------------------------------------------------------------------------
    /** Initialize the algorithm's properties.
     */
    void LoadILLReflectometry::init()
    {
      declareProperty(new FileProperty("Filename", "", FileProperty::Load, ".nxs"),
          "File path of the Data file to load");

      declareProperty(new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
          "The name to use for the output workspace");
    }

//----------------------------------------------------------------------------------------------
```

```
    /** Execute the algorithm.
     */
    void LoadILLReflectometry::exec()
    {
      // Retrieve filename
      std::string filenameData = getPropertyValue("Filename");

      // open the root node
      NeXus::NXRoot dataRoot(filenameData);
      NXEntry firstEntry = dataRoot.openFirstEntry();

      // Load Monitor details: n. monitors x monitor contents
      std::vector<std::vector<int> > monitorsData = loadMonitors(firstEntry);

      // Load Data details (number of tubes, channels, etc)
      loadDataDetails(firstEntry);

      std::string instrumentPath = m_loader.findInstrumentNexusPath(firstEntry);
      setInstrumentName(firstEntry, instrumentPath);

      initWorkSpace(firstEntry, monitorsData);

      g_log.debug("Building properties...");
      loadNexusEntriesIntoProperties(filenameData);

//      g_log.debug("Loading data...");
      loadDataIntoTheWorkSpace(firstEntry, monitorsData);

      // load the instrument from the IDF if it exists
      g_log.debug("Loading instrument definition...");
      runLoadInstrument();

      // 1) Move

      // Get distance and tilt angle stored in nexus file
      // Mantid way
      //        auto angleProp = dynamic_cast<PropertyWithValue<double>*>(m_localWorkspace->run().getProperty("dan.value"));
      // Nexus way
      double angle = firstEntry.getFloat("instrument/dan/value");       // detector angle in degrees
      double distance = firstEntry.getFloat("instrument/det/value");    // detector distance in millimeter
      distance /= 1000; // convert to meter
      placeDetector(distance, angle);

      // 2) Center, (must be done after move)
      int par1_101 = firstEntry.getInt("instrument/PSD/ny");
      g_log.debug("Note: using PSD/ny instead of PSD/nx. Should be corrected in next D17 nexus file.");
      double xCenter = 0.1325 / par1_101;       // As in lamp, but in meter
      centerDetector(xCenter);

      // Set the channel width property
      auto channel_width = dynamic_cast<PropertyWithValue<double>*>(m_localWorkspace->run().getProperty(
          "monitor1.time_of_flight_0"));
      m_localWorkspace->mutableRun().addProperty<double>("channel_width", *channel_width, true); //overwrite

      // Set the output workspace property
      setProperty("OutputWorkspace", m_localWorkspace);
    }

    /**
     * Set member variable with the instrument name
     */
    void LoadILLReflectometry::setInstrumentName(const NeXus::NXEntry &firstEntry,
        const std::string &instrumentNamePath)
    {

      if (instrumentNamePath == "")
      {
        std::string message("Cannot set the instrument name from the Nexus file!");
        g_log.error(message);
        throw std::runtime_error(message);
      }
      m_instrumentName = m_loader.getStringFromNexusPath(firstEntry, instrumentNamePath + "/name");
      boost::to_upper(m_instrumentName); // "D17" in file, keep it upper case.
      g_log.debug() << "Instrument name set to: " + m_instrumentName << std::endl;

    }

    /**
     * Creates the workspace and initialises member variables with
     * the corresponding values
     *
     * @param entry :: The Nexus entry
     * @param monitorsData :: Monitors data already loaded
     *
     */
    void LoadILLReflectometry::initWorkSpace(NeXus::NXEntry& /*entry*/,
        std::vector<std::vector<int> > monitorsData)
    {

      // dim0 * m_numberOfPixelsPerTube is the total number of detectors
      m_numberOfHistograms = m_numberOfTubes * m_numberOfPixelsPerTube;

      g_log.debug() << "NumberOfTubes: " << m_numberOfTubes << std::endl;
      g_log.debug() << "NumberOfPixelsPerTube: " << m_numberOfPixelsPerTube << std::endl;
      g_log.debug() << "NumberOfChannels: " << m_numberOfChannels << std::endl;
      g_log.debug() << "Monitors: " << monitorsData.size() << std::endl;
      g_log.debug() << "Monitors[0]: " << monitorsData[0].size() << std::endl;
      g_log.debug() << "Monitors[1]: " << monitorsData[1].size() << std::endl;

      // Now create the output workspace

      m_localWorkspace = WorkspaceFactory::Instance().create("Workspace2D",
          m_numberOfHistograms + monitorsData.size(), m_numberOfChannels + 1, m_numberOfChannels);

      m_localWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create("TOF");

      m_localWorkspace->setYUnitLabel("Counts");

    }

    /**
     * Load Data details (number of tubes, channels, etc)
     * @param entry First entry of nexus file
     */
    void LoadILLReflectometry::loadDataDetails(NeXus::NXEntry& entry)
```

```
    {
      // read in the data
      NXData dataGroup = entry.openNXData("data");
      NXInt data = dataGroup.openIntData();

      m_numberOfTubes = static_cast<size_t>(data.dim0());
      m_numberOfPixelsPerTube = static_cast<size_t>(data.dim1());
      m_numberOfChannels = static_cast<size_t>(data.dim2());


    }

    /**
     * Load monitors data found in nexus file
     *
     * @param entry :: The Nexus entry
     *
     */
    std::vector<std::vector<int> > LoadILLReflectometry::loadMonitors(NeXus::NXEntry& entry)
    {
      // read in the data
      g_log.debug("Fetching monitor data...");

      NXData dataGroup = entry.openNXData("monitor1/data");
      NXInt data = dataGroup.openIntData();
      // load the counts from the file into memory
      data.load();

      std::vector<std::vector<int> > monitors(1); // vector of monitors with one entry
      std::vector<int> monitor1(data(), data() + data.size());
      monitors[0].swap(monitor1);

      // There is two monitors in data file, but the second one seems to be always 0
      dataGroup = entry.openNXData("monitor2/data");
      data = dataGroup.openIntData();
      data.load();

      std::vector<int> monitor2(data(), data() + data.size());
      monitors.push_back(monitor2);

      return monitors;
    }

    /**
     * Load data found in nexus file
     *
     * @param entry :: The Nexus entry
     * @param monitorsData :: Monitors data already loaded
     *
     */
    void LoadILLReflectometry::loadDataIntoTheWorkSpace(NeXus::NXEntry& entry,
        std::vector<std::vector<int> > monitorsData)
    {

      m_wavelength = entry.getFloat("wavelength");
      double ei = m_loader.calculateEnergy(m_wavelength);
      m_localWorkspace->mutableRun().addProperty<double>("Ei", ei, true); //overwrite

      // read in the data
      NXData dataGroup = entry.openNXData("data");
      NXInt data = dataGroup.openIntData();
      // load the counts from the file into memory
      data.load();

      // Assign calculated bins to first X axis
////  m_localWorkspace->dataX(0).assign(detectorTofBins.begin(), detectorTofBins.end());

      size_t spec = 0;
      size_t nb_monitors = monitorsData.size();

      Progress progress(this, 0, 1, m_numberOfTubes * m_numberOfPixelsPerTube + nb_monitors);

      // Assign tof values to first X axis

      // 1) Get some parameters from nexus file and properties
      //    Note : This should be changed following future D17/ILL nexus file improvement.
      auto tof_channel_width_prop =
          dynamic_cast<PropertyWithValue<double>*>(m_localWorkspace->run().getProperty(
              "monitor1.time_of_flight_0"));
      m_channelWidth = *tof_channel_width_prop; /* PAR1[95] */

      auto tof_delay_prop = dynamic_cast<PropertyWithValue<double>*>(m_localWorkspace->run().getProperty(
          "monitor1.time_of_flight_2"));
      double tof_delay = *tof_delay_prop; /* PAR1[96] */

      double POFF = entry.getFloat("instrument/VirtualChopper/poff"); /* par1[54] */
      double mean_chop_2_phase = entry.getFloat("instrument/VirtualChopper/chopper2_phase_average"); /* PAR2[114] */

      //double mean_chop_1_phase = entry.getFloat("instrument/VirtualChopper/chopper1_phase_average"); /* PAR2[110] */
      // this entry seems to be wrong for now, we use the old one instead [YR 5/06/2014]
      double mean_chop_1_phase = entry.getFloat("instrument/Chopper1/phase");

      double open_offset = entry.getFloat("instrument/VirtualChopper/open_offset"); /* par1[56] */
      double chop1_speed = entry.getFloat("instrument/VirtualChopper/chopper1_speed_average"); /* PAR2[109] */


      g_log.debug() << "m_numberOfChannels: " << m_numberOfChannels << std::endl;
      g_log.debug() << "m_channelWidth: " << m_channelWidth << std::endl;
      g_log.debug() << "tof_delay: " << tof_delay << std::endl;
      g_log.debug() << "POFF: " << POFF << std::endl;
      g_log.debug() << "open_offset: " << open_offset << std::endl;
      g_log.debug() << "mean_chop_2_phase: " << mean_chop_2_phase << std::endl;
      g_log.debug() << "mean_chop_1_phase: " << mean_chop_1_phase << std::endl;
      g_log.debug() << "chop1_speed: " << chop1_speed << std::endl;

      // Thanks to Miguel Gonzales/ILL for this TOF formula
      double t_TOF2 = - 1.e6 * 60.0 * (POFF - 45.0 + mean_chop_2_phase - mean_chop_1_phase + open_offset)
                              /
                              (2.0 * 360 * chop1_speed);

      g_log.debug() << "t_TOF2: " << t_TOF2 << std::endl;

      // 2) Compute tof values
      for (size_t timechannelnumber = 0; timechannelnumber <= m_numberOfChannels; ++timechannelnumber)
      {
```

```
    double t_TOF1 = (static_cast<int>(timechannelnumber) + 0.5) * m_channelWidth + tof_delay;

    //g_log.debug() << "t_TOF1: " << t_TOF1 << std::endl;

    m_localWorkspace->dataX(0)[timechannelnumber] = t_TOF1 + t_TOF2 ;

  }


  // Load monitors
  for (size_t im = 0; im < nb_monitors; im++)
  {

    if (im > 0)
    {
      m_localWorkspace->dataX(im) = m_localWorkspace->readX(0);
    }

    // Assign Y
    int* monitor_p = monitorsData[im].data();
    m_localWorkspace->dataY(im).assign(monitor_p, monitor_p + m_numberOfChannels);

    progress.report();
  }

  // TODO
  // copy data if m_numberOfTubes = 1 or m_numberOfPixelsPerTube = 1

  // Then Tubes
  for (size_t i = 0; i < m_numberOfTubes; ++i)
  {
    for (size_t j = 0; j < m_numberOfPixelsPerTube; ++j)
    {

      // just copy the time binning axis to every spectra
      m_localWorkspace->dataX(spec + nb_monitors) = m_localWorkspace->readX(0);

      // Assign Y
      int* data_p = &data(static_cast<int>(i), static_cast<int>(j), 0);
      m_localWorkspace->dataY(spec + nb_monitors).assign(data_p, data_p + m_numberOfChannels);

      // Assign Error
      MantidVec& E = m_localWorkspace->dataE(spec + nb_monitors);
      std::transform(data_p, data_p + m_numberOfChannels, E.begin(),
          LoadHelper::calculateStandardError);

      ++spec;
      progress.report();
    }
  }  // for m_numberOfTubes

}  // LoadILLIndirect::loadDataIntoTheWorkSpace



void LoadILLReflectometry::loadNexusEntriesIntoProperties(std::string nexusfilename)
{

  API::Run & runDetails = m_localWorkspace->mutableRun();

  // Open NeXus file
  NXhandle nxfileID;
  NXstatus stat = NXopen(nexusfilename.c_str(), NXACC_READ, &nxfileID);
  if (stat == NX_ERROR)
  {
    g_log.debug() << "convertNexusToProperties: Error loading " << nexusfilename;
    throw Kernel::Exception::FileError("Unable to open File:", nexusfilename);
  }
  m_loader.addNexusFieldsToWsRun(nxfileID, runDetails);

  // Add also "Facility", as asked
  runDetails.addProperty("Facility", std::string("ILL"));

  stat = NXclose(&nxfileID);
}

/**
 * Run the Child Algorithm LoadInstrument.
 */
void LoadILLReflectometry::runLoadInstrument()
{

  IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");

  // Now execute the Child Algorithm. Catch and log any error, but don't stop.
  try
  {

    loadInst->setPropertyValue("InstrumentName", m_instrumentName);
    loadInst->setProperty<MatrixWorkspace_sptr>("Workspace", m_localWorkspace);
    loadInst->execute();

  } catch (...)
  {
    g_log.information("Cannot load the instrument definition.");
  }
}

void LoadILLReflectometry::centerDetector(double xCenter)
{

  std::string componentName("uniq_detector");
  V3D pos = m_loader.getComponentPosition(m_localWorkspace, componentName);
  // TODO confirm!
  pos.setX(pos.X() - xCenter);
  m_loader.moveComponent(m_localWorkspace, componentName, pos);

}

void LoadILLReflectometry::placeDetector(double distance /* meter */, double angle /* degree */)
{

  std::string componentName("uniq_detector");
  V3D pos = m_loader.getComponentPosition(m_localWorkspace, componentName);
```

```
      double r, theta, phi;
      pos.getSpherical(r, theta, phi);

      V3D newpos;
      newpos.spherical(distance, angle, phi);

      m_loader.moveComponent(m_localWorkspace, componentName, newpos);

      // Apply a local rotation to stay perpendicular to the beam
      const V3D axis(0.0, 1.0, 0.0);
      Quat rotation(angle, axis);
      m_loader.rotateComponent(m_localWorkspace, componentName, rotation);

    }
  } // namespace DataHandling
} // namespace Mantid
```

LoadILLSANS	Loads a ILL nexus files for SANS instruments.	ILL: D33

```
#include "MantidDataHandling/LoadILLSANS.h"
#include "MantidAPI/FileProperty.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidKernel/UnitFactory.h"

#include <limits>
#include <numeric>      // std::accumulate

namespace Mantid {
namespace DataHandling {

using namespace Kernel;
using namespace API;
using namespace NeXus;

DECLARE_NEXUS_FILELOADER_ALGORITHM(LoadILLSANS)

//----------------------------------------------------------------------------------------------
/** Constructor
 */
LoadILLSANS::LoadILLSANS() :
m_defaultBinning(2)
{
        m_supportedInstruments.push_back("D33");
}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
LoadILLSANS::~LoadILLSANS() {

}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string LoadILLSANS::name() const {
        return "LoadILLSANS";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int LoadILLSANS::version() const {
        return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string LoadILLSANS::category() const {
        return "DataHandling";
}

//----------------------------------------------------------------------------------------------


/**
 * Return the confidence with with this algorithm can load the file
 * @param descriptor A descriptor for the file
 * @returns An integer specifying the confidence level. 0 indicates it will not be used
 */
int LoadILLSANS::confidence(Kernel::NexusDescriptor & descriptor) const {
        // fields existent only at the ILL for SANS machines
        if (descriptor.pathExists("/entry0/reactor_power")
                        && descriptor.pathExists("/entry0/instrument_name")
                        && descriptor.pathExists("/entry0/mode")) {
                return 80;
        } else {
                return 0;
        }
}


//----------------------------------------------------------------------------------------------
/** Initialize the algorithm's properties.
 */
void LoadILLSANS::init() {
        declareProperty(
                        new FileProperty("Filename", "", FileProperty::Load, ".nxs"),
                        "Name of the SPE file to load");
        declareProperty(
                        new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
                        "The name to use for the output workspace");
}

//----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void LoadILLSANS::exec() {
        // Init
        std::string filename = getPropertyValue("Filename");
        NXRoot root(filename);
        NXEntry firstEntry = root.openFirstEntry();

        std::string instrumentPath = m_loader.findInstrumentNexusPath(firstEntry);
        setInstrumentName(firstEntry, instrumentPath);

        g_log.debug("Setting detector positions...");
        DetectorPosition detPos = getDetectorPosition(firstEntry, instrumentPath);

        initWorkSpace(firstEntry, instrumentPath);


        // load the instrument from the IDF if it exists
        runLoadInstrument();

        // Move detectors
        moveDetectors(detPos);
```

```
        setFinalProperties();
        // Set the output workspace property
        setProperty("OutputWorkspace", m_localWorkspace);

}

/**
 * Set member variable with the instrument name
 */
void LoadILLSANS::setInstrumentName(const NeXus::NXEntry &firstEntry,
                const std::string &instrumentNamePath) {

        if (instrumentNamePath == "") {
                std::string message(
                                "Cannot set the instrument name from the Nexus file!");
                g_log.error(message);
                throw std::runtime_error(message);
        }
        m_instrumentName = m_loader.getStringFromNexusPath(firstEntry,
                        instrumentNamePath + "/name");
        g_log.debug() << "Instrument name set to: " + m_instrumentName << std::endl;

}

/**
 * Get detector panel distances from the nexus file
 * @return a structure with the positions
 */
DetectorPosition LoadILLSANS::getDetectorPosition(
                const NeXus::NXEntry &firstEntry,
                const std::string &instrumentNamePath) {
        std::string detectorPath(instrumentNamePath + "/detector");

        DetectorPosition pos;

        pos.distanceSampleRear = m_loader.getDoubleFromNexusPath(firstEntry,
                        detectorPath + "/det2_calc");
        pos.distanceSampleBottomTop = m_loader.getDoubleFromNexusPath(firstEntry,
                        detectorPath + "/det1_calc");
        pos.distanceSampleRightLeft = pos.distanceSampleBottomTop
                        + m_loader.getDoubleFromNexusPath(firstEntry,
                                        detectorPath + "/det1_panel_separation");

        pos.shiftLeft = m_loader.getDoubleFromNexusPath(firstEntry,
                        detectorPath + "/OxL_actual") * 1e-3;
        pos.shiftRight = m_loader.getDoubleFromNexusPath(firstEntry,
                        detectorPath + "/OxR_actual") * 1e-3;
        pos.shiftUp = m_loader.getDoubleFromNexusPath(firstEntry,
                        detectorPath + "/OyT_actual") * 1e-3;
        pos.shiftDown = m_loader.getDoubleFromNexusPath(firstEntry,
                        detectorPath + "/OyB_actual") * 1e-3;

        g_log.debug() << pos;

        return pos;
}

void LoadILLSANS::initWorkSpace(NeXus::NXEntry &firstEntry,
                const std::string &instrumentPath) {

        g_log.debug("Fetching data...");

        NXData dataGroup1 = firstEntry.openNXData("data1");
        NXInt dataRear = dataGroup1.openIntData();
        dataRear.load();
        NXData dataGroup2 = firstEntry.openNXData("data2");
        NXInt dataRight = dataGroup2.openIntData();
        dataRight.load();
        NXData dataGroup3 = firstEntry.openNXData("data3");
        NXInt dataLeft = dataGroup3.openIntData();
        dataLeft.load();
        NXData dataGroup4 = firstEntry.openNXData("data4");
        NXInt dataDown = dataGroup4.openIntData();
        dataDown.load();
        NXData dataGroup5 = firstEntry.openNXData("data5");
        NXInt dataUp = dataGroup5.openIntData();
        dataUp.load();
        g_log.debug("Checking channel numbers...");

        // check number of channels
        if (dataRear.dim2() != dataRight.dim2()
                        && dataRight.dim2() != dataLeft.dim2()
                        && dataLeft.dim2() != dataDown.dim2()
                        && dataDown.dim2() != dataUp.dim2()) {
                throw std::runtime_error(
                                "The time bins have not the same dimension for all the 5 detectors!");
        }
        int numberOfHistograms = dataRear.dim0() * dataRear.dim1()
                        + dataRight.dim0() * dataRight.dim1()
                        + dataLeft.dim0() * dataLeft.dim1()
                        + dataDown.dim0() * dataDown.dim1() + dataUp.dim0() * dataUp.dim1();

        g_log.debug("Creating empty workspace...");
        // TODO : Must put this 2 somewhere else: number of monitors!
        createEmptyWorkspace(numberOfHistograms+2, dataRear.dim2());

        loadMetaData(firstEntry, instrumentPath);


        std::vector<double> binningRear, binningRight, binningLeft, binningDown,binningUp;

        if (firstEntry.getFloat("mode") == 0.0) { // Not TOF
                g_log.debug("Getting default wavelength bins...");
                binningRear = m_defaultBinning;
                binningRight = m_defaultBinning;
                binningLeft = m_defaultBinning;
                binningDown = m_defaultBinning;
                binningUp = m_defaultBinning;

        }
        else {
                g_log.debug("Getting wavelength bins from the nexus file...");
                std::string binPathPrefix(
                                instrumentPath + "/tof/tof_wavelength_detector");
```

```
                binningRear = m_loader.getTimeBinningFromNexusPath(firstEntry,
                                binPathPrefix + "1");
                binningRight = m_loader.getTimeBinningFromNexusPath(firstEntry,
                                binPathPrefix + "2");
                binningLeft = m_loader.getTimeBinningFromNexusPath(firstEntry,
                                binPathPrefix + "3");
                binningDown = m_loader.getTimeBinningFromNexusPath(firstEntry,
                                binPathPrefix + "4");
                binningUp = m_loader.getTimeBinningFromNexusPath(firstEntry,
                                binPathPrefix + "5");
        }
        g_log.debug("Loading the data into the workspace...");
        size_t nextIndex = loadDataIntoWorkspaceFromMonitors(firstEntry,0);
        nextIndex = loadDataIntoWorkspaceFromHorizontalTubes(dataRear,binningRear,nextIndex);
        nextIndex = loadDataIntoWorkspaceFromVerticalTubes(dataRight,binningRight,nextIndex);
        nextIndex = loadDataIntoWorkspaceFromVerticalTubes(dataLeft,binningLeft,nextIndex);
        nextIndex = loadDataIntoWorkspaceFromHorizontalTubes(dataDown,binningDown,nextIndex);
        nextIndex = loadDataIntoWorkspaceFromHorizontalTubes(dataUp,binningUp,nextIndex);
}

size_t LoadILLSANS::loadDataIntoWorkspaceFromMonitors(NeXus::NXEntry &firstEntry, size_t firstIndex) {

        // let's find the monitors
        // For D33 should be monitor1 and monitor2
        for (std::vector<NXClassInfo>::const_iterator it =
                        firstEntry.groups().begin(); it != firstEntry.groups().end(); ++it) {
                if (it->nxclass == "NXmonitor") {
                        NXData dataGroup = firstEntry.openNXData(it->nxname);
                        NXInt data = dataGroup.openIntData();
                        data.load();
                        g_log.debug() << "Monitor: " << it->nxname << " dims = " << data.dim0() << "x"<< data.dim1() << "x"<< data.dim2() <<
std::endl;

                        const size_t vectorSize = data.dim2() + 1;
                        std::vector<double> positionsBinning;
                        positionsBinning.reserve(vectorSize);

                        for( size_t i = 0; i < vectorSize; i++ )
                                positionsBinning.push_back( static_cast<double>(i) );

                        // Assign X
                        m_localWorkspace->dataX(firstIndex).assign(positionsBinning.begin(),positionsBinning.end());
                        // Assign Y
                        m_localWorkspace->dataY(firstIndex).assign(data(), data() + data.dim2());
                        // Assign Error
                        MantidVec& E = m_localWorkspace->dataE(firstIndex);
                        std::transform(data(), data() + data.dim2(), E.begin(),LoadHelper::calculateStandardError);

                        // Add average monitor counts to a property:
                        double averageMonitorCounts = std::accumulate(data(), data() + data.dim2(), 0) / data.dim2();
                        // make sure the monitor has values!
                        if (averageMonitorCounts > 0) {
                                API::Run & runDetails = m_localWorkspace->mutableRun();
                                runDetails.addProperty("monitor", averageMonitorCounts,true);
                        }

                        firstIndex++;
                }
        }
        return firstIndex;
}

size_t LoadILLSANS::loadDataIntoWorkspaceFromHorizontalTubes(NeXus::NXInt &data,
                const std::vector<double> &timeBinning, size_t firstIndex = 0) {

        g_log.debug("Loading the data into the workspace:");
        g_log.debug() << "\t" << "firstIndex = " << firstIndex << std::endl;
        g_log.debug() << "\t" << "Number of Pixels : data.dim0() = " << data.dim0() << std::endl;
        g_log.debug() << "\t" << "Number of Tubes : data.dim1() = " << data.dim1() << std::endl;
        g_log.debug() << "\t" << "data.dim2() = " << data.dim2() << std::endl;
        g_log.debug() << "\t" << "First bin = " << timeBinning[0] << std::endl;


        // Workaround to get the number of tubes / pixels
        size_t numberOfTubes = data.dim1();
        size_t numberOfPixelsPerTube  = data.dim0();

        Progress progress(this, 0, 1, data.dim0() * data.dim1());

        m_localWorkspace->dataX(firstIndex).assign(timeBinning.begin(),timeBinning.end());

        size_t spec = firstIndex;
        for (size_t i = 0; i < numberOfTubes; ++i) { // iterate tubes
                for (size_t j = 0; j < numberOfPixelsPerTube; ++j) { // iterate pixels in the tube 256
                        if (spec > firstIndex) {
                                // just copy the time binning axis to every spectra
                                m_localWorkspace->dataX(spec) = m_localWorkspace->readX(firstIndex);
                        }
                        // Assign Y
                        int* data_p =&data(static_cast<int>(j), static_cast<int>(i), 0);
                        m_localWorkspace->dataY(spec).assign(data_p, data_p + data.dim2());

                        // Assign Error
                        MantidVec& E = m_localWorkspace->dataE(spec);
                        std::transform(data_p, data_p + data.dim2(), E.begin(),LoadHelper::calculateStandardError);

                        ++spec;
                        progress.report();
                }
        }

        g_log.debug() << "Data loading into WS done...." << std::endl;

        return spec;
}


size_t LoadILLSANS::loadDataIntoWorkspaceFromVerticalTubes(NeXus::NXInt &data,
                const std::vector<double> &timeBinning, size_t firstIndex = 0) {

        g_log.debug("Loading the data into the workspace:");
        g_log.debug() << "\t" << "firstIndex = " << firstIndex << std::endl;
        g_log.debug() << "\t" << "Number of Tubes : data.dim0() = " << data.dim0() << std::endl;
        g_log.debug() << "\t" << "Number of Pixels : data.dim1() = " << data.dim1() << std::endl;
        g_log.debug() << "\t" << "data.dim2() = " << data.dim2() << std::endl;
```

```
        g_log.debug() << "\t" << "First bin = " << timeBinning[0] << std::endl;


        // Workaround to get the number of tubes / pixels
        size_t numberOfTubes = data.dim0();
        size_t numberOfPixelsPerTube  = data.dim1();

        Progress progress(this, 0, 1, data.dim0() * data.dim1());

        m_localWorkspace->dataX(firstIndex).assign(timeBinning.begin(),timeBinning.end());

        size_t spec = firstIndex;
        for (size_t i = 0; i < numberOfTubes; ++i) { // iterate tubes
                for (size_t j = 0; j < numberOfPixelsPerTube; ++j) { // iterate pixels in the tube 256
                        if (spec > firstIndex) {
                                // just copy the time binning axis to every spectra
                                m_localWorkspace->dataX(spec) = m_localWorkspace->readX(firstIndex);
                        }
                        // Assign Y
                        int* data_p =&data(static_cast<int>(i), static_cast<int>(j), 0);
                        m_localWorkspace->dataY(spec).assign(data_p, data_p + data.dim2());

                        // Assign Error
                        MantidVec& E = m_localWorkspace->dataE(spec);
                        std::transform(data_p, data_p + data.dim2(), E.begin(),LoadHelper::calculateStandardError);

                        ++spec;
                        progress.report();
                }
        }

        g_log.debug() << "Data loading inti WS done...." << std::endl;

        return spec;
}

/***
 * Create a workspace without any data in it
 */
void LoadILLSANS::createEmptyWorkspace(int numberOfHistograms,
                int numberOfChannels) {
        m_localWorkspace = WorkspaceFactory::Instance().create("Workspace2D",
                        numberOfHistograms, numberOfChannels + 1, numberOfChannels);
        m_localWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create(
                        "Wavelength");
        m_localWorkspace->setYUnitLabel("Counts");
}


void LoadILLSANS::runLoadInstrument() {

        IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");

        // Now execute the Child Algorithm. Catch and log any error, but don't stop.
        try {
                loadInst->setPropertyValue("InstrumentName", m_instrumentName);
                loadInst->setProperty<MatrixWorkspace_sptr>("Workspace",
                                m_localWorkspace);
                loadInst->execute();
        } catch (...) {
                g_log.information("Cannot load the instrument definition.");
        }
}

void LoadILLSANS::moveDetectors(const DetectorPosition& detPos) {

        // Move in Z
        moveDetectorDistance(detPos.distanceSampleRear, "back_detector");
        moveDetectorDistance(detPos.distanceSampleBottomTop, "front_detector_top");
        moveDetectorDistance(detPos.distanceSampleBottomTop, "front_detector_bottom");
        moveDetectorDistance(detPos.distanceSampleRightLeft, "front_detector_right");
        moveDetectorDistance(detPos.distanceSampleRightLeft, "front_detector_left");
        //Move in X
        moveDetectorHorizontal(detPos.shiftLeft,"front_detector_left");
        moveDetectorHorizontal(-detPos.shiftRight,"front_detector_right");
        //Move in Y
        moveDetectorVertical(detPos.shiftUp,"front_detector_top");
        moveDetectorVertical(-detPos.shiftDown,"front_detector_bottom");
}

/**
 * Move detectors in Z axis (X,Y are kept constant)
 */
void LoadILLSANS::moveDetectorDistance(double distance, const std::string& componentName) {

        API::IAlgorithm_sptr mover = createChildAlgorithm(
                        "MoveInstrumentComponent");
        V3D pos = getComponentPosition(componentName);
        try {
                mover->setProperty<MatrixWorkspace_sptr>("Workspace", m_localWorkspace);
                mover->setProperty("ComponentName", componentName);
                mover->setProperty("X", pos.X());
                mover->setProperty("Y", pos.Y());
                mover->setProperty("Z", distance);
                mover->setProperty("RelativePosition", false);
                mover->executeAsChildAlg();
                g_log.debug() << "Moving component '" << componentName << "' to Z = " << distance << std::endl;
        } catch (std::exception &e) {
                g_log.error() << "Cannot move the component '" << componentName << "' to Z = " << distance << std::endl;
                g_log.error() << e.what() << std::endl;
        }
}

/**
 * Move detectors in X
 */
void LoadILLSANS::moveDetectorHorizontal(double shift, const std::string& componentName) {

        API::IAlgorithm_sptr mover = createChildAlgorithm(
                        "MoveInstrumentComponent");
        V3D pos = getComponentPosition(componentName);
        try {
                mover->setProperty<MatrixWorkspace_sptr>("Workspace", m_localWorkspace);
                mover->setProperty("ComponentName", componentName);
                mover->setProperty("X", shift);
```

```
                mover->setProperty("Y", pos.Y());
                mover->setProperty("Z", pos.Z());
                mover->setProperty("RelativePosition", false);
                mover->executeAsChildAlg();
                g_log.debug() << "Moving component '" << componentName << "' to X = " << shift << std::endl;
        } catch (std::exception &e) {
                g_log.error() << "Cannot move the component '" << componentName << "' to X = " << shift << std::endl;
                g_log.error() << e.what() << std::endl;
        }
}

void LoadILLSANS::moveDetectorVertical(double shift, const std::string& componentName) {

        API::IAlgorithm_sptr mover = createChildAlgorithm(
                        "MoveInstrumentComponent");
        V3D pos = getComponentPosition(componentName);
        try {
                mover->setProperty<MatrixWorkspace_sptr>("Workspace", m_localWorkspace);
                mover->setProperty("ComponentName", componentName);
                mover->setProperty("X", pos.X());
                mover->setProperty("Y", shift);
                mover->setProperty("Z", pos.Z());
                mover->setProperty("RelativePosition", false);
                mover->executeAsChildAlg();
                g_log.debug() << "Moving component '" << componentName << "' to Y = " << shift << std::endl;
        } catch (std::exception &e) {
                g_log.error() << "Cannot move the component '" << componentName << "' to Y = " << shift << std::endl;
                g_log.error() << e.what() << std::endl;
        }
}

/**
 * Get position in space of a componentName
 */
V3D LoadILLSANS::getComponentPosition(const std::string& componentName) {
        Geometry::Instrument_const_sptr instrument = m_localWorkspace->getInstrument();
        Geometry::IComponent_const_sptr component = instrument->getComponentByName(componentName);
        return component->getPos();
}

/*
 * Loads metadata present in the nexus file
 */
void LoadILLSANS::loadMetaData(const NeXus::NXEntry &entry, const std::string &instrumentNamePath) {

        g_log.debug("Loading metadata...");

        API::Run & runDetails = m_localWorkspace->mutableRun();

        int runNum = entry.getInt("run_number");
        std::string run_num = boost::lexical_cast<std::string>(runNum);
        runDetails.addProperty("run_number", run_num);

        if (entry.getFloat("mode") == 0.0) { // Not TOF
                runDetails.addProperty<std::string>("tof_mode", "Non TOF");
        }
        else{
                runDetails.addProperty<std::string>("tof_mode", "TOF");
        }

        std::string desc = m_loader.getStringFromNexusPath(entry,"sample_description");
        runDetails.addProperty("sample_description", desc);

        std::string start_time = entry.getString("start_time");
        start_time = m_loader.dateTimeInIsoFormat(start_time);
        runDetails.addProperty("run_start", start_time);

        std::string end_time = entry.getString("end_time");
        end_time = m_loader.dateTimeInIsoFormat(end_time);
        runDetails.addProperty("run_end", end_time);

    double duration = entry.getFloat("duration");
    runDetails.addProperty("timer", duration);

        double wavelength = entry.getFloat(instrumentNamePath + "/selector/wavelength");
        g_log.debug()<< "Wavelength found in the nexus file: " << wavelength << std::endl;

        if (wavelength <= 0) {
                g_log.debug()<< "Mode = " << entry.getFloat("mode") << std::endl;
                g_log.information("The wavelength present in the NeXus file <= 0.");
                if (entry.getFloat("mode") == 0.0) { // Not TOF
                        throw std::runtime_error("Working in Non TOF mode and the wavelength in the file is <=0 !!! Check with the instrument
scientist!");
                }
        }
        else {
                double wavelengthRes = entry.getFloat(instrumentNamePath + "/selector/wavelength_res");
                runDetails.addProperty<double>("wavelength", wavelength);
                double ei = m_loader.calculateEnergy(wavelength);
                runDetails.addProperty<double>("Ei", ei, true);
                // wavelength
                m_defaultBinning[0] = wavelength - wavelengthRes * wavelength * 0.01 / 2;
                m_defaultBinning[1] = wavelength + wavelengthRes * wavelength * 0.01 / 2;
        }

        // Put the detector distances:
//      std::string detectorPath(instrumentNamePath + "/detector");
//      // Just for Sample - RearDetector
//      double sampleDetectorDistance = m_loader.getDoubleFromNexusPath(entry,detectorPath + "/det2_calc");
//      runDetails.addProperty("sample_detector_distance", sampleDetectorDistance);



}


/**
 * @param lambda : wavelength in Amstrongs
 * @param twoTheta : twoTheta in degreess
 */
double LoadILLSANS::calculateQ(const double lambda, const double twoTheta) const
{
        return (4 * 3.1415936 * std::sin(twoTheta*(3.1415936/180)/2)) / (lambda);

}
```

```
std::pair<double, double> LoadILLSANS::calculateQMaxQMin(){
        double min= std::numeric_limits<double>::max(), max= std::numeric_limits<double>::min();
        g_log.debug("Calculating Qmin Qmax...");
        std::size_t nHist = m_localWorkspace->getNumberHistograms();
        for (std::size_t i=0; i < nHist; ++i){
                Geometry::IDetector_const_sptr det = m_localWorkspace->getDetector(i);
                if ( ! det->isMonitor() ){
                        const MantidVec& lambdaBinning = m_localWorkspace->readX(i);
                        Kernel::V3D detPos = det->getPos();
                        double r, theta, phi;
                        detPos.getSpherical(r, theta, phi);
                        double v1 = calculateQ(*(lambdaBinning.begin()),theta);
                        double v2 = calculateQ(*(lambdaBinning.end()-1),theta);
                        //std::cout << "i=" << i << " theta="<<theta << " lambda_i=" << *(lambdaBinning.begin()) << " lambda_f=" <<
*(lambdaBinning.end()-1) << " v1=" << v1 << " v2=" << v2 << std::endl;
                        if ( i == 0) {
                                min = v1;
                                max = v1;
                        }
                        if (v1 < min){
                                min = v1;
                        }
                        if (v2 < min){
                                min = v2;
                        }
                        if (v1 > max){
                                max = v1;
                        }
                        if (v2 > max){
                                max = v2;
                        }
                }
                else
                        g_log.debug() << "Detector " << i << " is a Monitor : " << det->getID() << std::endl;
        }

        g_log.debug() << "Calculating Qmin Qmax. Done : [" << min << "," << max <<"]"<< std::endl;

        return std::pair<double, double>(min,max);
}

void LoadILLSANS::setFinalProperties(){
        API::Run & runDetails = m_localWorkspace->mutableRun();
        runDetails.addProperty("is_frame_skipping", 0);

        std::pair<double, double> minmax = LoadILLSANS::calculateQMaxQMin();
        runDetails.addProperty("qmin", minmax.first);
        runDetails.addProperty("qmax", minmax.second);
}


} // namespace DataHandling
} // namespace Mantid
```

LoadLLB	Loads LLB nexus file.	LLB: MiBemol

```
#include "MantidDataHandling/LoadLLB.h"
#include "MantidAPI/FileProperty.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidAPI/Progress.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidGeometry/Instrument.h"

#include <limits>
#include <algorithm>
#include <iostream>
#include <vector>
#include <cmath>

namespace Mantid {
namespace DataHandling {

using namespace Kernel;
using namespace API;
using namespace NeXus;

DECLARE_NEXUS_FILELOADER_ALGORITHM(LoadLLB);


//----------------------------------------------------------------------------------------------
/** Constructor
 */
LoadLLB::LoadLLB() {
        m_instrumentName = "";
        m_supportedInstruments.push_back("MIBEMOL");
}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
LoadLLB::~LoadLLB() {
}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string LoadLLB::name() const {
        return "LoadLLB";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int LoadLLB::version() const {
        return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string LoadLLB::category() const {
        return "DataHandling";
}

/**
 * Return the confidence with with this algorithm can load the file
 * @param descriptor A descriptor for the file
 * @returns An integer specifying the confidence level. 0 indicates it will not be used
 */
int LoadLLB::confidence(Kernel::NexusDescriptor & descriptor) const {
        // fields existent only at the LLB
        if (descriptor.pathExists("/nxentry/program_name")
                        && descriptor.pathExists("/nxentry/subrun_number")
                        && descriptor.pathExists("/nxentry/total_subruns")) {
                return 80;
        } else {
                return 0;
        }
}

//----------------------------------------------------------------------------------------------

//----------------------------------------------------------------------------------------------
/** Initialize the algorithm's properties.
 */
void LoadLLB::init() {
        std::vector<std::string> exts;
        exts.push_back(".nxs");
        exts.push_back(".hdf");
        declareProperty(new FileProperty("Filename", "", FileProperty::Load, exts),
                        "The name of the Nexus file to load");
        declareProperty(
                        new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
                        "The name to use for the output workspace");
}

//----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void LoadLLB::exec() {

        std::string filename = getPropertyValue("Filename");
        NXRoot root(filename);
        NXEntry entry = root.openFirstEntry();
        setInstrumentName(entry);

        initWorkSpace(entry);
        runLoadInstrument(); // just to get IDF
        loadTimeDetails(entry);
        loadDataIntoTheWorkSpace(entry);

        loadRunDetails(entry);
        loadExperimentDetails(entry);

        runLoadInstrument();
```

```
        setProperty("OutputWorkspace", m_localWorkspace);
}

void LoadLLB::setInstrumentName(NeXus::NXEntry& entry) {

        m_instrumentPath = "nxinstrument";
        m_instrumentName = m_loader.getStringFromNexusPath(entry, m_instrumentPath + "/name");

        if (m_instrumentName == "") {
                throw std::runtime_error("Cannot read the instrument name from the Nexus file!");
        }
        g_log.debug() << "Instrument Name: " << m_instrumentName
                                                << " in NxPath: " << m_instrumentPath << std::endl;
}

void LoadLLB::initWorkSpace(NeXus::NXEntry& entry) {

        // read in the data
        NXData dataGroup = entry.openNXData("nxdata");
        NXInt data = dataGroup.openIntData();

        m_numberOfTubes = static_cast<size_t>(data.dim0());
        m_numberOfPixelsPerTube = 1;
        m_numberOfChannels = static_cast<size_t>(data.dim1());

        // dim0 * m_numberOfPixelsPerTube is the total number of detectors
        m_numberOfHistograms = m_numberOfTubes * m_numberOfPixelsPerTube;

        g_log.debug() << "NumberOfTubes: " << m_numberOfTubes << std::endl;
        g_log.debug() << "NumberOfPixelsPerTube: " << m_numberOfPixelsPerTube
                        << std::endl;
        g_log.debug() << "NumberOfChannels: " << m_numberOfChannels << std::endl;

        // Now create the output workspace
        // Might need to get this value from the number of monitors in the Nexus file
        // params:
        // workspace type,
        // total number of spectra + (number of monitors = 0),
        // bin boundaries = m_numberOfChannels + 1
        // Z/time dimension
        m_localWorkspace = WorkspaceFactory::Instance().create("Workspace2D",
                        m_numberOfHistograms, m_numberOfChannels + 1, m_numberOfChannels);
        m_localWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create(
                        "TOF");
        m_localWorkspace->setYUnitLabel("Counts");

}

/**
 *
 */
void LoadLLB::loadTimeDetails(NeXus::NXEntry& entry) {

        m_wavelength = entry.getFloat("nxbeam/incident_wavelength");
        // Apparently this is in the wrong units
        // http://iramis.cea.fr/Phocea/file.php?class=page&reload=1227895533&file=21/How_to_install_and_use_the_Fitmib_suite_v28112008.pdf
        m_channelWidth = entry.getInt("nxmonitor/channel_width") * 0.1;

        g_log.debug("Nexus Data:");
        g_log.debug() << " ChannelWidth: " << m_channelWidth << std::endl;
        g_log.debug() << " Wavelength: " << m_wavelength << std::endl;

}

void LoadLLB::loadDataIntoTheWorkSpace(NeXus::NXEntry& entry) {

        // read in the data
        NXData dataGroup = entry.openNXData("nxdata");
        NXFloat data = dataGroup.openFloatData();
        data.load();

        // EPP
        int calculatedDetectorElasticPeakPosition = getDetectorElasticPeakPosition(
                        data);

        std::vector<double> timeBinning = getTimeBinning(
                        calculatedDetectorElasticPeakPosition, m_channelWidth);

        // Assign time bin to first X entry
        m_localWorkspace->dataX(0).assign(timeBinning.begin(), timeBinning.end());

        Progress progress(this, 0, 1, m_numberOfTubes * m_numberOfPixelsPerTube);
        size_t spec = 0;
        for (size_t i = 0; i < m_numberOfTubes; ++i) {
                for (size_t j = 0; j < m_numberOfPixelsPerTube; ++j) {
                        if (spec > 0) {
                                // just copy the time binning axis to every spectra
                                m_localWorkspace->dataX(spec) = m_localWorkspace->readX(0);
                        }
                        // Assign Y
                        float* data_p = &data(static_cast<int>(i), static_cast<int>(j));
                        m_localWorkspace->dataY(spec).assign(data_p,
                                        data_p + m_numberOfChannels);

                        // Assign Error
                        MantidVec& E = m_localWorkspace->dataE(spec);
                        std::transform(data_p, data_p + m_numberOfChannels, E.begin(),
                                        LoadLLB::calculateError);

                        ++spec;
                        progress.report();
                }
        }

        g_log.debug() << "Data loading inti WS done...." << std::endl;
}

int LoadLLB::getDetectorElasticPeakPosition(const NeXus::NXFloat &data) {

        std::vector<int> cumulatedSumOfSpectras(m_numberOfChannels, 0);
        for (size_t i = 0; i < m_numberOfTubes; i++)
        {
                float* data_p = &data(static_cast<int>(i), 0);
                float currentSpec = 0;
```

```
                for (size_t j = 0; j  < m_numberOfChannels; ++j)
                        currentSpec += data_p[j];

                if(i > 0)
                {
                        cumulatedSumOfSpectras[i] = cumulatedSumOfSpectras[i-1] + static_cast<int>(currentSpec);
                }
                else
                {
                        cumulatedSumOfSpectras[i] = static_cast<int>(currentSpec);
                }
        }
        auto it = std::max_element(cumulatedSumOfSpectras.begin(),
                        cumulatedSumOfSpectras.end());

        int calculatedDetectorElasticPeakPosition;
        if (it == cumulatedSumOfSpectras.end()) {
                throw std::runtime_error(
                                "No Elastic peak position found while analyzing the data!");
        } else {
                //calculatedDetectorElasticPeakPosition = *it;
                calculatedDetectorElasticPeakPosition = static_cast<int>(std::distance(
                                cumulatedSumOfSpectras.begin(), it));

                if (calculatedDetectorElasticPeakPosition == 0) {
                        throw std::runtime_error(
                                        "No Elastic peak position found while analyzing the data. Elastic peak position is ZERO!");
                } else {
                        g_log.debug() << "Calculated Detector EPP: "
                                        << calculatedDetectorElasticPeakPosition << std::endl;
                }
        }
        return calculatedDetectorElasticPeakPosition;
}

std::vector<double> LoadLLB::getTimeBinning(int elasticPeakPosition,
                double channelWidth) {

        double l1 = m_loader.getL1(m_localWorkspace);
        double l2 = m_loader.getL2(m_localWorkspace);

        double theoreticalElasticTOF = (m_loader.calculateTOF(l1,m_wavelength) + m_loader.calculateTOF(l2,m_wavelength)) * 1e6; //microsecs

        g_log.debug() << "elasticPeakPosition : "
                        << static_cast<float>(elasticPeakPosition) << std::endl;
        g_log.debug() << "l1 : " << l1 << std::endl;
        g_log.debug() << "l2 : " << l2 << std::endl;
        g_log.debug() << "theoreticalElasticTOF : " << theoreticalElasticTOF
                        << std::endl;

        std::vector<double> detectorTofBins(m_numberOfChannels + 1);

        for (size_t i = 0; i < m_numberOfChannels + 1; ++i) {
                detectorTofBins[i] = theoreticalElasticTOF
                                + channelWidth
                                                * static_cast<double>(static_cast<int>(i)
                                                                - elasticPeakPosition) - channelWidth / 2; // to make sure the bin is in the
middle of the elastic peak

        }
        return detectorTofBins;
}


void LoadLLB::loadRunDetails(NXEntry & entry) {

        API::Run & runDetails = m_localWorkspace->mutableRun();

//      int runNum = entry.getInt("run_number");
//      std::string run_num = boost::lexical_cast<std::string>(runNum);
//      runDetails.addProperty("run_number", run_num);

        std::string start_time = entry.getString("start_time");
        //start_time = getDateTimeInIsoFormat(start_time);
        runDetails.addProperty("run_start", start_time);

        std::string end_time = entry.getString("end_time");
        //end_time = getDateTimeInIsoFormat(end_time);
        runDetails.addProperty("run_end", end_time);

        double wavelength = entry.getFloat("nxbeam/incident_wavelength");
        runDetails.addProperty<double>("wavelength", wavelength);

        double energy = m_loader.calculateEnergy(wavelength);
        runDetails.addProperty<double>("Ei", energy, true); //overwrite

        std::string title = entry.getString("title");
        runDetails.addProperty("title", title);
        m_localWorkspace->setTitle(title);

}

/*
 * Load data about the Experiment.
 *
 * TODO: This is very incomplete. In ISIS they much more info in the nexus file than ILL.
 *
 * @param entry :: The Nexus entry
 */
void LoadLLB::loadExperimentDetails(NXEntry & entry) {

        // TODO: Do the rest
        // Pick out the geometry information

        (void) entry;

//      std::string description = boost::lexical_cast<std::string>(
//                      entry.getFloat("sample/description"));
//
//      m_localWorkspace->mutableSample().setName(description);

//      m_localWorkspace->mutableSample().setThickness(static_cast<double> (isis_raw->spb.e_thick));
//      m_localWorkspace->mutableSample().setHeight(static_cast<double> (isis_raw->spb.e_height));
//      m_localWorkspace->mutableSample().setWidth(static_cast<double> (isis_raw->spb.e_width));
```

```
}

/**
 * Run the Child Algorithm LoadInstrument.
 */
void LoadLLB::runLoadInstrument() {

        IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");

        // Now execute the Child Algorithm. Catch and log any error, but don't stop.
        try {

                // TODO: depending on the m_numberOfPixelsPerTube we might need to load a different IDF

                loadInst->setPropertyValue("InstrumentName", m_instrumentName);
                loadInst->setProperty<MatrixWorkspace_sptr>("Workspace",
                                m_localWorkspace);
                loadInst->execute();
        } catch (...) {
                g_log.information("Cannot load the instrument definition.");
        }
}

} // namespace DataHandling
} // namespace Mantid
```

LoadSINQFocus → LoadSINQ	Loads a FOCUS nexus file from the PSI	SINQ: FOCUS

```
#include "MantidDataHandling/LoadSINQFocus.h"
#include "MantidAPI/FileProperty.h"
#include "MantidAPI/Progress.h"
#include "MantidAPI/RegisterFileLoader.h"
#include "MantidGeometry/Instrument.h"
#include "MantidKernel/UnitFactory.h"

#include <limits>
#include <algorithm>
#include <iostream>
#include <vector>
#include <cmath>

namespace Mantid {
namespace DataHandling {

using namespace Kernel;
using namespace API;
using namespace NeXus;

DECLARE_NEXUS_FILELOADER_ALGORITHM(LoadSINQFocus);

//----------------------------------------------------------------------------------------------
/** Constructor
 */
LoadSINQFocus::LoadSINQFocus() {
        m_instrumentName = "";
        m_supportedInstruments.push_back("FOCUS");
    this->useAlgorithm("LoadSINQ");
    this->deprecatedDate("2013-10-28");
}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
LoadSINQFocus::~LoadSINQFocus() {
}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string LoadSINQFocus::name() const {
        return "LoadSINQFocus";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int LoadSINQFocus::version() const {
        return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string LoadSINQFocus::category() const {
        return "DataHandling";
}

//----------------------------------------------------------------------------------------------

/**
 * Return the confidence with with this algorithm can load the file
 * @param descriptor A descriptor for the file
 * @returns An integer specifying the confidence level. 0 indicates it will not be used
 */
int LoadSINQFocus::confidence(Kernel::NexusDescriptor & descriptor) const {

    // fields existent only at the SINQ (to date Loader only valid for focus)
    if (descriptor.pathExists("/entry1/FOCUS/SINQ") ){
            return 80;
    } else {
            return 0;
    }
}

//----------------------------------------1----------------------------------------------------
/** Initialize the algorithm's properties.
 */
void LoadSINQFocus::init() {
        std::vector<std::string> exts;
        exts.push_back(".nxs");
        exts.push_back(".hdf");
        declareProperty(new FileProperty("Filename", "", FileProperty::Load, exts),
                        "The name of the Nexus file to load");
        declareProperty(
                        new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
                        "The name to use for the output workspace");
}

//----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void LoadSINQFocus::exec() {

        std::string filename = getPropertyValue("Filename");
        NXRoot root(filename);
        NXEntry entry = root.openFirstEntry();
        setInstrumentName(entry);

        initWorkSpace(entry);

        loadDataIntoTheWorkSpace(entry);

        loadRunDetails(entry);
        loadExperimentDetails(entry);

        runLoadInstrument();
```

```
        setProperty("OutputWorkspace", m_localWorkspace);
}

/*
 * Set global variables:
 * m_instrumentPath
 * m_instrumentName
 * Note that the instrument in the nexus file is of the form "FOCUS at SINQ"
 *
 */
void LoadSINQFocus::setInstrumentName(NeXus::NXEntry& entry) {

        m_instrumentPath = m_loader.findInstrumentNexusPath(entry);

        if (m_instrumentPath == "") {
                throw std::runtime_error("Cannot set the instrument name from the Nexus file!");
        }
        m_instrumentName = m_loader.getStringFromNexusPath(entry, m_instrumentPath + "/name");
        size_t pos = m_instrumentName.find(" ");
        m_instrumentName = m_instrumentName.substr(0, pos);
}

void LoadSINQFocus::initWorkSpace(NeXus::NXEntry& entry) {

        // read in the data
        NXData dataGroup = entry.openNXData("merged");
        NXInt data = dataGroup.openIntData();

        m_numberOfTubes = static_cast<size_t>(data.dim0());
        m_numberOfPixelsPerTube = 1;
        m_numberOfChannels = static_cast<size_t>(data.dim1());

        // dim0 * m_numberOfPixelsPerTube is the total number of detectors
        m_numberOfHistograms = m_numberOfTubes * m_numberOfPixelsPerTube;

        g_log.debug() << "NumberOfTubes: " << m_numberOfTubes << std::endl;
        g_log.debug() << "NumberOfPixelsPerTube: " << m_numberOfPixelsPerTube
                      << std::endl;
        g_log.debug() << "NumberOfChannels: " << m_numberOfChannels << std::endl;

        // Now create the output workspace
        // Might need to get this value from the number of monitors in the Nexus file
        // params:
        // workspace type,
        // total number of spectra + (number of monitors = 0),
        // bin boundaries = m_numberOfChannels + 1
        // Z/time dimension
        m_localWorkspace = WorkspaceFactory::Instance().create("Workspace2D",
                        m_numberOfHistograms, m_numberOfChannels + 1, m_numberOfChannels);
        m_localWorkspace->getAxis(0)->unit() = UnitFactory::Instance().create(
                        "TOF");
        m_localWorkspace->setYUnitLabel("Counts");

}

void LoadSINQFocus::loadDataIntoTheWorkSpace(NeXus::NXEntry& entry) {

        // read in the data
        NXData dataGroup = entry.openNXData("merged");
        NXInt data = dataGroup.openIntData();
        data.load();

        std::vector<double> timeBinning = m_loader.getTimeBinningFromNexusPath(entry, "merged/time_binning");
        m_localWorkspace->dataX(0).assign(timeBinning.begin(),timeBinning.end());

        Progress progress(this, 0, 1, m_numberOfTubes * m_numberOfPixelsPerTube);
        size_t spec = 0;
        for (size_t i = 0; i < m_numberOfTubes; ++i) {
                for (size_t j = 0; j < m_numberOfPixelsPerTube; ++j) {
                        if (spec > 0) {
                                // just copy the time binning axis to every spectra
                                m_localWorkspace->dataX(spec) = m_localWorkspace->readX(0);
                        }
                        // Assign Y
                        int* data_p = &data(static_cast<int>(i), static_cast<int>(j));
                        m_localWorkspace->dataY(spec).assign(data_p,
                                        data_p + m_numberOfChannels);
                        // Assign Error
                        MantidVec& E = m_localWorkspace->dataE(spec);
                        std::transform(data_p, data_p + m_numberOfChannels, E.begin(),
                                        LoadSINQFocus::calculateError);
                        ++spec;
                        progress.report();
                }
        }
        g_log.debug() << "Data loading into WS done...." << std::endl;
}

void LoadSINQFocus::loadRunDetails(NXEntry & entry) {

        API::Run & runDetails = m_localWorkspace->mutableRun();

//      int runNum = entry.getInt("run_number");
//      std::string run_num = boost::lexical_cast<std::string>(runNum);
//      runDetails.addProperty("run_number", run_num);

        std::string start_time = entry.getString("start_time");
        //start_time = getDateTimeInIsoFormat(start_time);
        runDetails.addProperty("run_start", start_time);

        std::string end_time = entry.getString("end_time");
        //end_time = getDateTimeInIsoFormat(end_time);
        runDetails.addProperty("run_end", end_time);

        double wavelength = entry.getFloat(m_instrumentPath + "/monochromator/lambda");
        runDetails.addProperty<double>("wavelength", wavelength);

        double energy = entry.getFloat(m_instrumentPath + "/monochromator/energy");
        runDetails.addProperty<double>("Ei", energy, true); //overwrite

        std::string title = entry.getString("title");
        runDetails.addProperty("title", title);
        m_localWorkspace->setTitle(title);
```

```
}

/*
 * Load data about the Experiment.
 *
 * TODO: This is very incomplete. In ISIS they much more info in the nexus file than ILL.
 *
 * @param entry :: The Nexus entry
 */
void LoadSINQFocus::loadExperimentDetails(NXEntry & entry) {

        std::string name = boost::lexical_cast<std::string>(
                        entry.getFloat("sample/name"));
        m_localWorkspace->mutableSample().setName(name);

}

/**
 * Run the Child Algorithm LoadInstrument.
 */
void LoadSINQFocus::runLoadInstrument() {

        IAlgorithm_sptr loadInst = createChildAlgorithm("LoadInstrument");

        // Now execute the Child Algorithm. Catch and log any error, but don't stop.
        try {

                // TODO: depending on the m_numberOfPixelsPerTube we might need to load a different IDF

                loadInst->setPropertyValue("InstrumentName", m_instrumentName);
                loadInst->setProperty<MatrixWorkspace_sptr>("Workspace",
                                m_localWorkspace);
                loadInst->execute();
        } catch (...) {
                g_log.information("Cannot load the instrument definition.");
        }
}

} // namespace DataHandling
} // namespace Mantid
```

ConvertEmptyToTof	Converts the channel number to time of flight.	TOF (ILL)

```
//----------------------------------------------------------------------
// Includes
//----------------------------------------------------------------------
#include "MantidAlgorithms/ConvertEmptyToTof.h"

#include "MantidAPI/WorkspaceValidators.h"
#include "MantidKernel/ArrayProperty.h"
#include "MantidAPI/FunctionFactory.h"
#include "MantidAPI/IPeakFunction.h"
#include "MantidAPI/ConstraintFactory.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidKernel/BoundedValidator.h"

#include <cmath>
#include <map>
#include <numeric>      // std::accumulate
#include <utility>      // std::pair

namespace Mantid {
namespace Algorithms {

using namespace Kernel;
using namespace API;

// Register the algorithm into the AlgorithmFactory
DECLARE_ALGORITHM(ConvertEmptyToTof)

//----------------------------------------------------------------------------------------------
/** Constructor
 */
ConvertEmptyToTof::ConvertEmptyToTof() {

}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
ConvertEmptyToTof::~ConvertEmptyToTof() {
}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string ConvertEmptyToTof::name() const {
  return "ConvertEmptyToTof";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int ConvertEmptyToTof::version() const {
  return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string ConvertEmptyToTof::category() const {
  return "Transforms\\Units";
}


//----------------------------------------------------------------------------------------------
/** Initialize the algorithm's properties.
 */
void ConvertEmptyToTof::init() {

  auto wsValidator = boost::make_shared<CompositeValidator>();
  wsValidator->add<WorkspaceUnitValidator>("Empty");
  declareProperty(
      new WorkspaceProperty<DataObjects::Workspace2D>("InputWorkspace", "",
          Direction::Input, wsValidator), "Name of the input workspace");
  declareProperty(
      new WorkspaceProperty<API::MatrixWorkspace>("OutputWorkspace", "",
          Direction::Output),
      "Name of the output workspace, can be the same as the input");
  declareProperty(new Kernel::ArrayProperty<int>("ListOfSpectraIndices"),
      "A list of spectra indices as a string with ranges; e.g. 5-10,15,20-23. \n"
          "Optional: if not specified, then the Start/EndIndex fields are used alone. "
          "If specified, the range and the list are combined (without duplicating indices). For example, a range of 10 to 20 and a list
'12,15,26,28' gives '10-20,26,28'.");

  declareProperty(new Kernel::ArrayProperty<int>("ListOfChannelIndices"),
      "A list of spectra indices as a string with ranges; e.g. 5-10,15,20-23. \n"
          "Optional: if not specified, then the Start/EndIndex fields are used alone. "
          "If specified, the range and the list are combined (without duplicating indices). For example, a range of 10 to 20 and a list
'12,15,26,28' gives '10-20,26,28'.");

  // OR Specify EPP
  auto mustBePositive = boost::make_shared<BoundedValidator<int> >();
  mustBePositive->setLower(0);
  declareProperty("ElasticPeakPosition", EMPTY_INT(), mustBePositive,
      "Value of elastic peak position if none of the above are filled in.");
  declareProperty("ElasticPeakPositionSpectrum", EMPTY_INT(), mustBePositive,
      "Spectrum index used for elastic peak position above.");

}

//----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void ConvertEmptyToTof::exec() {

  m_inputWS = this->getProperty("InputWorkspace");
  m_outputWS = this->getProperty("OutputWorkspace");
  std::vector<int> spectraIndices = getProperty("ListOfSpectraIndices");
  std::vector<int> channelIndices = getProperty("ListOfChannelIndices");
  int epp = getProperty("ElasticPeakPosition");
  int eppSpectrum = getProperty("ElasticPeakPositionSpectrum");
```

```
  std::vector<double> tofAxis;
  double channelWidth = getPropertyFromRun<double>(m_inputWS,"channel_width");

  // If the ElasticPeakPosition and the ElasticPeakPositionSpectrum were specified
  if (epp != EMPTY_INT() && eppSpectrum != EMPTY_INT()) {
    g_log.information(
        "Using the specified ElasticPeakPosition and ElasticPeakPositionSpectrum");

    double wavelength = getPropertyFromRun<double>(m_inputWS, "wavelength");
    double l1 = getL1(m_inputWS);
    double l2 = getL2(m_inputWS, eppSpectrum);
    double epTof = (calculateTOF(l1, wavelength) + calculateTOF(l2, wavelength)) * 1e6; //microsecs

    tofAxis = makeTofAxis(epp, epTof,m_inputWS->blocksize() + 1, channelWidth);

  }
  // If the spectraIndices and channelIndices were specified
  else {

    //validations
    validateSpectraIndices(spectraIndices);
    validateChannelIndices(channelIndices);

    //Map of spectra index, epp
    std::map<int, int> eppMap = findElasticPeakPositions(spectraIndices,
        channelIndices);

    for (auto it = eppMap.begin(); it != eppMap.end(); ++it) {
      g_log.debug() << "Spectra idx =" << it->first << ", epp=" << it->second
          << std::endl;
    }

    std::pair<int, double> eppAndEpTof = findAverageEppAndEpTof(eppMap);

    tofAxis = makeTofAxis(eppAndEpTof.first, eppAndEpTof.second,
        m_inputWS->blocksize() + 1, channelWidth);
  }

  // If input and output workspaces are not the same, create a new workspace for the output
  if (m_outputWS != m_inputWS) {
    m_outputWS = API::WorkspaceFactory::Instance().create(m_inputWS);
  }

  setTofInWS(tofAxis, m_outputWS);

  setProperty("OutputWorkspace", m_outputWS);

}

/**
 * Check if spectra indices are in the limits of the number of histograms
 * in the input workspace. If v is empty, uses all spectra.
 * @param v :: vector with the spectra indices
 */
void ConvertEmptyToTof::validateSpectraIndices(std::vector<int> &v) {
  auto nHist = m_inputWS->getNumberHistograms();
  if (v.size() == 0) {
    g_log.information(
        "No spectrum index given. Using all spectra to calculate the elastic peak.");
    // use all spectra indices
    v.reserve(nHist);
    for (unsigned int i = 0; i < nHist; ++i)
      v[i] = i;
  } else {
    for (auto it = v.begin(); it != v.end(); ++it) {
      if (*it < 0 || static_cast<size_t>(*it) >= nHist) {
        throw std::runtime_error(
            "Spectra index out of limits: "
                + boost::lexical_cast<std::string>(*it));
      }
    }
  }
}

/**
 * Check if the channel indices are in the limits of the number of the block size
 * in the input workspace. If v is empty, uses all channels.
 * @param v :: vector with the channel indices to use
 */
void ConvertEmptyToTof::validateChannelIndices(std::vector<int> &v) {
  auto blockSize = m_inputWS->blocksize() + 1;
  if (v.size() == 0) {
    g_log.information(
        "No channel index given. Using all channels (full spectrum!) to calculate the elastic peak.");
    // use all channel indices
    v.reserve(blockSize);
    for (unsigned int i = 0; i < blockSize; ++i)
      v[i] = i;
  } else {
    for (auto it = v.begin(); it != v.end(); ++it) {
      if (*it < 0 || static_cast<size_t>(*it) >= blockSize) {
        throw std::runtime_error(
            "Channel index out of limits: "
                + boost::lexical_cast<std::string>(*it));
      }
    }
  }
}

/**
 * Looks for the EPP positions in the spectraIndices
 * @return map with worskpace spectra index, elastic peak position for this spectra
 */
std::map<int, int> ConvertEmptyToTof::findElasticPeakPositions(
    const std::vector<int> &spectraIndices,
    const std::vector<int> &channelIndices) {

  std::map<int, int> eppMap;

  // make sure we not looking for channel indices outside the bounds
  assert(
      static_cast<size_t>(*(channelIndices.end() - 1))
          < m_inputWS->blocksize() + 1);

  g_log.information() << "Peak detection, search for peak " << std::endl;
```

```
    for (auto it = spectraIndices.begin(); it != spectraIndices.end(); ++it) {

      int spectrumIndex = *it;
      const Mantid::MantidVec& thisSpecY = m_inputWS->dataY(spectrumIndex);

      int minChannelIndex = *(channelIndices.begin());
      int maxChannelIndex = *(channelIndices.end() - 1);

      double center, sigma, height, minX, maxX;
      minX = static_cast<double>(minChannelIndex);
      maxX = static_cast<double>(maxChannelIndex);
      estimateFWHM(thisSpecY, center, sigma, height, minX, maxX);

      g_log.debug() << "Peak estimate :: center=" << center << "\t sigma="
          << sigma << "\t height=" << height << "\t minX=" << minX << "\t maxX="
          << maxX << std::endl;

      bool doFit = doFitGaussianPeak(spectrumIndex, center, sigma, height, minX,
          maxX);
      if (!doFit) {
        g_log.error() << "doFitGaussianPeak failed..." << std::endl;
        throw std::runtime_error("Gaussin Peak Fit failed....");
      }

      g_log.debug() << "Peak Fitting :: center=" << center << "\t sigma=" << sigma
          << "\t height=" << height << "\t minX=" << minX << "\t maxX=" << maxX
          << std::endl;

      // round up the center to the closest int
      eppMap[spectrumIndex] = roundUp(center);

    }
    return eppMap;

  }

  /**
   * Estimated the FWHM for Gaussian peak fitting
   *
   */
  void ConvertEmptyToTof::estimateFWHM(const Mantid::MantidVec& spec,
      double& center, double& sigma, double& height, double& minX, double& maxX) {

    auto maxValueIt = std::max_element(spec.begin() + static_cast<size_t>(minX),
        spec.begin() + static_cast<size_t>(maxX)); // max value
    double maxValue = *maxValueIt;
    size_t maxIndex = std::distance(spec.begin(), maxValueIt); // index of max value

    //indices and values for the fwhm detection
    size_t minFwhmIndex = maxIndex;
    size_t maxFwhmIndex = maxIndex;
    double minFwhmValue = maxValue;
    double maxFwhmValue = maxValue;
    // fwhm detection
    for (; minFwhmValue > 0.5 * maxValue; minFwhmIndex--, minFwhmValue =
        spec[minFwhmIndex]) {
    }
    for (; maxFwhmValue > 0.5 * maxValue; maxFwhmIndex++, maxFwhmValue =
        spec[maxFwhmIndex]) {
    }
    //double fwhm = thisSpecX[maxFwhmIndex] - thisSpecX[minFwhmIndex + 1];
    double fwhm = static_cast<double>(maxFwhmIndex - minFwhmIndex + 1);

    //parameters for the gaussian peak fit
    center = static_cast<double>(maxIndex);
    sigma = fwhm;
    height = maxValue;

    g_log.debug() << "Peak estimate  : center=" << center << "\t sigma=" << sigma
        << "\t h=" << height << std::endl;

    //determination of the range used for the peak definition
    size_t ipeak_min = std::max(static_cast<size_t>(0),
        maxIndex
            - static_cast<size_t>(2.5
                * static_cast<double>(maxIndex - maxFwhmIndex)));
    size_t ipeak_max = std::min(spec.size(),
        maxIndex
            + static_cast<size_t>(2.5
                * static_cast<double>(maxFwhmIndex - maxIndex)));
    size_t i_delta_peak = ipeak_max - ipeak_min;

    g_log.debug() << "Peak estimate xmin/max: " << ipeak_min - 1 << "\t"
        << ipeak_max + 1 << std::endl;

    minX = static_cast<double>(ipeak_min - 2 * i_delta_peak);
    maxX = static_cast<double>(ipeak_max + 2 * i_delta_peak);

  }

  /**
   * Fit peak without background i.e, with background removed
   *  inspired from FitPowderDiffPeaks.cpp
   *  copied from PoldiPeakDetection2.cpp
   *
   @param workspaceindex :: indice of the row to use
   @param center :: gaussian parameter - center
   @param sigma :: gaussian parameter - width
   @param height :: gaussian parameter - height
   @param startX :: fit range - start X value
   @param endX :: fit range - end X value
   @returns A boolean status flag, true for fit success, false else
   */
  bool ConvertEmptyToTof::doFitGaussianPeak(int workspaceindex, double& center,
      double& sigma, double& height, double startX, double endX) {

    g_log.debug("Calling doFitGaussianPeak...");

    // 1. Estimate
    sigma = sigma * 0.5;

    // 2. Use factory to generate Gaussian
    auto temppeak = API::FunctionFactory::Instance().createFunction("Gaussian");
    auto gaussianpeak = boost::dynamic_pointer_cast<API::IPeakFunction>(temppeak);
```

```
  gaussianpeak->setHeight(height);
  gaussianpeak->setCentre(center);
  gaussianpeak->setFwhm(sigma);

  // 3. Constraint
  double centerleftend = center - sigma * 0.5;
  double centerrightend = center + sigma * 0.5;
  std::ostringstream os;
  os << centerleftend << " < PeakCentre < " << centerrightend;
  auto * centerbound = API::ConstraintFactory::Instance().createInitialized(
      gaussianpeak.get(), os.str(), false);
  gaussianpeak->addConstraint(centerbound);

  g_log.debug("Calling createChildAlgorithm : Fit...");
  // 4. Fit
  API::IAlgorithm_sptr fitalg = createChildAlgorithm("Fit", -1, -1, true);
  fitalg->initialize();

  fitalg->setProperty("Function",
      boost::dynamic_pointer_cast<API::IFunction>(gaussianpeak));
  fitalg->setProperty("InputWorkspace", m_inputWS);
  fitalg->setProperty("WorkspaceIndex", workspaceindex);
  fitalg->setProperty("Minimizer", "Levenberg-MarquardtMD");
  fitalg->setProperty("CostFunction", "Least squares");
  fitalg->setProperty("MaxIterations", 1000);
  fitalg->setProperty("Output", "FitGaussianPeak");
  fitalg->setProperty("StartX", startX);
  fitalg->setProperty("EndX", endX);

  // 5.  Result
  bool successfulfit = fitalg->execute();
  if (!fitalg->isExecuted() || !successfulfit) {
    // Early return due to bad fit
    g_log.warning() << "Fitting Gaussian peak for peak around "
        << gaussianpeak->centre() << std::endl;
    return false;
  }

  // 6. Get result
  center = gaussianpeak->centre();
  height = gaussianpeak->height();
  double fwhm = gaussianpeak->fwhm();
  if (fwhm <= 0.0) {
    return false;
  }
//    sigma = fwhm*2;
  //  sigma = fwhm/2.35;

  return true;
}

/**
 * Finds the TOF for a given epp
 * @param eppMap : pair workspace spec index - epp
 * @return the average EPP and the corresponding average EP in TOF
 */

std::pair<int, double> ConvertEmptyToTof::findAverageEppAndEpTof(
    const std::map<int, int>& eppMap) {

  double l1 = getL1(m_inputWS);
  double wavelength = getPropertyFromRun<double>(m_inputWS, "wavelength");

  std::vector<double> epTofList;
  std::vector<int> eppList;

  double firstL2 = getL2(m_inputWS, eppMap.begin()->first);
  for (auto it = eppMap.begin(); it != eppMap.end(); ++it) {

    double l2 = getL2(m_inputWS, it->first);
    if (!areEqual(l2, firstL2, 0.0001)) {
      g_log.error() << "firstL2=" << firstL2 << " , " << "l2=" << l2
          << std::endl;
      throw std::runtime_error(
          "All the pixels for selected spectra must have the same distance from the sample!");
    } else {
      firstL2 = l2;
    }

    epTofList.push_back(
        (calculateTOF(l1, wavelength) + calculateTOF(l2, wavelength)) * 1e6); //microsecs
    eppList.push_back(it->first);

    g_log.debug() << "WS index = " << it->first << ", l1 = " << l1 << ", l2 = "
        << l2 << ", TOF(l1+l2) = " << *(epTofList.end() - 1) << std::endl;
  }

  double averageEpTof = std::accumulate(epTofList.begin(), epTofList.end(), 0.0)
      / static_cast<double>(epTofList.size());
  int averageEpp = roundUp(
      static_cast<double>(std::accumulate(eppList.begin(), eppList.end(), 0))
          / static_cast<double>(eppList.size()));

  g_log.debug() << "Average epp=" << averageEpp << " , Average epTof="
      << averageEpTof << std::endl;
  return std::make_pair(averageEpp, averageEpTof);
}

double ConvertEmptyToTof::getL1(API::MatrixWorkspace_const_sptr workspace) {
  Geometry::Instrument_const_sptr instrument = workspace->getInstrument();
  Geometry::IComponent_const_sptr sample = instrument->getSample();
  double l1 = instrument->getSource()->getDistance(*sample);
  return l1;
}

double ConvertEmptyToTof::getL2(API::MatrixWorkspace_const_sptr workspace,
    int detId) {
  // Get a pointer to the instrument contained in the workspace
  Geometry::Instrument_const_sptr instrument = workspace->getInstrument();
  // Get the distance between the source and the sample (assume in metres)
  Geometry::IComponent_const_sptr sample = instrument->getSample();
  // Get the sample-detector distance for this detector (in metres)
  double l2 = workspace->getDetector(detId)->getPos().distance(
      sample->getPos());
  return l2;
```

```
}

double ConvertEmptyToTof::calculateTOF(double distance, double wavelength) {
  if (wavelength <= 0) {
    throw std::runtime_error("Wavelenght is <= 0");
  }

  double velocity = PhysicalConstants::h
      / (PhysicalConstants::NeutronMass * wavelength * 1e-10); //m/s

  return distance / velocity;
}

/**
 * Compare two double with a precision epsilon
 */
bool ConvertEmptyToTof::areEqual(double a, double b, double epsilon) {
  return fabs(a - b) < epsilon;
}

template<typename T>
T ConvertEmptyToTof::getPropertyFromRun(API::MatrixWorkspace_const_sptr inputWS,
    const std::string& propertyName) {
  if (inputWS->run().hasProperty(propertyName)) {
    Kernel::Property* prop = inputWS->run().getProperty(propertyName);
    return boost::lexical_cast<T>(prop->value());
  } else {
    std::string mesg = "No '" + propertyName
        + "' property found in the input workspace....";
    throw std::runtime_error(mesg);
  }
}

int ConvertEmptyToTof::roundUp(double value) {
  return static_cast<int>(std::floor(value + 0.5));
}

/**
 * Builds the X time axis
 */
std::vector<double> ConvertEmptyToTof::makeTofAxis(int epp, double epTof,
    size_t size, double channelWidth) {
  std::vector<double> axis(size);

  g_log.debug() << "Building the TOF X Axis: epp=" << epp << ", epTof=" << epTof
      << ", Channel Width=" << channelWidth << std::endl;
  for (size_t i = 0; i < size; ++i) {
    axis[i] = epTof
        + channelWidth * static_cast<double>(static_cast<int>(i) - epp)
        - channelWidth / 2; // to make sure the bin is in the middle of the elastic peak
  }
  g_log.debug() << "TOF X Axis: [start,end] = [" << *axis.begin() << ","
      << *(axis.end() - 1) << "]" << std::endl;
  return axis;
}

void ConvertEmptyToTof::setTofInWS(const std::vector<double> &tofAxis,
    API::MatrixWorkspace_sptr outputWS) {

  const size_t numberOfSpectra = m_inputWS->getNumberHistograms();
  int64_t numberOfSpectraInt64 = static_cast<int64_t>(numberOfSpectra); // cast to make openmp happy

  g_log.debug() << "Setting the TOF X Axis for numberOfSpectra="
      << numberOfSpectra << std::endl;

  Progress prog(this, 0.0, 0.2, numberOfSpectra);
  PARALLEL_FOR2(m_inputWS,outputWS)
  for (int64_t i = 0; i < numberOfSpectraInt64; ++i) {
    PARALLEL_START_INTERUPT_REGION
    // Just copy over
    outputWS->dataY(i) = m_inputWS->readY(i);
    outputWS->dataE(i) = m_inputWS->readE(i);
    // copy
    outputWS->setX(i, tofAxis);

    prog.report();
  PARALLEL_END_INTERUPT_REGION
}       //end for i
PARALLEL_CHECK_INTERUPT_REGION
outputWS->getAxis(0)->unit() = UnitFactory::Instance().create("TOF");

}

} // namespace Algorithms
} // namespace Mantid
```

CorrectFlightPaths	Used to correct flight paths in 2D shaped detectors.	TOF (ILL: IN5)

```
//----------------------------------------------------------------------
// Includes
//----------------------------------------------------------------------
#include "MantidAlgorithms/CorrectFlightPaths.h"
#include "MantidAPI/WorkspaceValidators.h"
#include "MantidDataObjects/Workspace2D.h"
#include "MantidDataObjects/EventWorkspace.h"
#include "MantidKernel/UnitFactory.h"
#include "MantidKernel/BoundedValidator.h"
#include "MantidKernel/ListValidator.h"
#include "MantidGeometry/Instrument/ComponentHelper.h"
#include "MantidGeometry/Instrument/ParameterMap.h"

#include <cmath>

namespace Mantid {
namespace Algorithms {

using namespace Kernel;
using namespace API;
using namespace Geometry;

// Register the class into the algorithm factory
DECLARE_ALGORITHM(CorrectFlightPaths)

// Constructor
CorrectFlightPaths::CorrectFlightPaths() :
    API::Algorithm() {
}

/** Initialisation method. Declares properties to be used in algorithm.
 *
 */
void CorrectFlightPaths::init() {

//todo: add validator for TOF

  auto wsValidator = boost::make_shared<CompositeValidator>();
  wsValidator->add<WorkspaceUnitValidator>("TOF");
  wsValidator->add<HistogramValidator>();
  declareProperty(
      new WorkspaceProperty<API::MatrixWorkspace>("InputWorkspace", "",
          Direction::Input, wsValidator), "Name of the input workspace");
  declareProperty(
      new WorkspaceProperty<API::MatrixWorkspace>("OutputWorkspace", "",
          Direction::Output),
      "Name of the output workspace, can be the same as the input");

}

/**
 * Initialises input and output workspaces.
 *
 */
void CorrectFlightPaths::initWorkspaces() {
  // Get the workspaces
  m_inputWS = this->getProperty("InputWorkspace");
  m_outputWS = this->getProperty("OutputWorkspace");
  m_instrument = m_inputWS->getInstrument();
  m_sample = m_instrument->getSample();
  // If input and output workspaces are not the same, create a new workspace for the output
  if (m_outputWS != this->m_inputWS) {
    m_outputWS = API::WorkspaceFactory::Instance().create(m_inputWS);
  }

  m_wavelength = getRunProperty("wavelength");
  g_log.debug() << "Wavelength = " << m_wavelength;
  m_l2 = getInstrumentProperty("l2");
  g_log.debug() << " L2 = " << m_l2 << std::endl;

}

/**
 * Executes the algorithm
 *
 */
void CorrectFlightPaths::exec() {

  initWorkspaces();

  Geometry::ParameterMap& pmap = m_outputWS->instrumentParameters();

  const size_t numberOfChannels = this->m_inputWS->blocksize();
// Calculate the number of spectra in this workspace
  const int numberOfSpectra = static_cast<int>(this->m_inputWS->size()
      / numberOfChannels);
  API::Progress prog(this, 0.0, 1.0, numberOfSpectra);

  int64_t numberOfSpectra_i = static_cast<int64_t>(numberOfSpectra); // cast to make openmp happy

  // Loop over the histograms (detector spectra)

  PARALLEL_FOR2(m_inputWS,m_outputWS)
  for (int64_t i = 0; i < numberOfSpectra_i; ++i) {
    //for (int64_t i = 32000; i < 32256; ++i) {
    PARALLEL_START_INTERUPT_REGION

    MantidVec& xOut = m_outputWS->dataX(i);
    MantidVec& yOut = m_outputWS->dataY(i);
    MantidVec& eOut = m_outputWS->dataE(i);
    const MantidVec& xIn = m_inputWS->readX(i);
    const MantidVec& yIn = m_inputWS->readY(i);
    const MantidVec& eIn = m_inputWS->readE(i);
    //Copy the energy transfer axis
    // TOF
//              MantidVec& xOut = m_outputWS->dataX(i);
//              const MantidVec& xIn = m_inputWS->readX(i);
```

```
    // subract the diference in l2
    IDetector_const_sptr det = m_inputWS->getDetector(i);
    double thisDetL2 = det->getDistance(*m_sample);
    //if (!det->isMonitor() && thisDetL2 != m_l2) {
    double deltaL2 = std::abs(thisDetL2 - m_l2);
    double deltaTOF = calculateTOF(deltaL2);
    deltaTOF *= 1e6; //micro sec

    // position - set all detector distance to constant l2
    double r, theta, phi;
    V3D oldPos = det->getPos();
    oldPos.getSpherical(r, theta, phi);
    V3D newPos;
    newPos.spherical(m_l2, theta, phi);
    ComponentHelper::moveComponent(*det, pmap, newPos,
        ComponentHelper::Absolute);

    unsigned int j = 0;
    for (; j < numberOfChannels; ++j) {
      xOut[j] = xIn[j] - deltaTOF;
      // there's probably a better way of copying this....
      yOut[j] = yIn[j];
      eOut[j] = eIn[j];

    }
    // last bin
    xOut[numberOfChannels] = xIn[numberOfChannels] + deltaTOF;
    //}
    prog.report("Aligning elastic line...");
  PARALLEL_END_INTERUPT_REGION
} //end for i
PARALLEL_CHECK_INTERUPT_REGION

this->setProperty("OutputWorkspace", this->m_outputWS);

}

/*
 * Get run property as double
 * @s - input property name
 *
 */
double CorrectFlightPaths::getRunProperty(std::string s) {
Mantid::Kernel::Property* prop = m_inputWS->run().getProperty(s);
double val;
if (!prop || !Strings::convert(prop->value(), val)) {
  std::string mesg = "Run property " + s + "doesn't exist!";
  g_log.error(mesg);
  throw std::runtime_error(mesg);
}
return val;
}
/*
 * Get instrument property as double
 * @s - input property name
 *
 */
double CorrectFlightPaths::getInstrumentProperty(std::string s) {
std::vector<std::string> prop = m_instrument->getStringParameter(s);
if (prop.empty()) {
  std::string mesg = "Property <" + s + "> doesn't exist!";
  g_log.error(mesg);
  throw std::runtime_error(mesg);
}
g_log.debug() << "prop[0] = " << prop[0] << std::endl;
return boost::lexical_cast<double>(prop[0]);
}

/*
 * Returns the neutron TOF
 * @distance - Distance in meters
 */
double CorrectFlightPaths::calculateTOF(double distance) {
double velocity = PhysicalConstants::h
    / (PhysicalConstants::NeutronMass * m_wavelength * 1e-10); //m/s

return distance / velocity;
}

} // namespace Algorithm
} // namespace Mantid
```

DetectorEfficiencyCorUser	This algorithm calculates the detector efficiency according the formula set in the instrument definition file/parameters.	TOF (ILL: IN4, IN5, IN6)

```
#include "MantidAlgorithms/DetectorEfficiencyCorUser.h"
#include "MantidAPI/WorkspaceValidators.h"
#include "MantidKernel/BoundedValidator.h"
#include "MantidKernel/CompositeValidator.h"
#include "MantidGeometry/muParser_Silent.h"
#include <ctime>

namespace Mantid {
namespace Algorithms {

using namespace Kernel;
using namespace API;
using namespace Geometry;

// Register the algorithm into the AlgorithmFactory
DECLARE_ALGORITHM(DetectorEfficiencyCorUser)

//----------------------------------------------------------------------------------------------
/** Constructor
 */
DetectorEfficiencyCorUser::DetectorEfficiencyCorUser() {
}

//----------------------------------------------------------------------------------------------
/** Destructor
 */
DetectorEfficiencyCorUser::~DetectorEfficiencyCorUser() {
}

//----------------------------------------------------------------------------------------------
/// Algorithm's name for identification. @see Algorithm::name
const std::string DetectorEfficiencyCorUser::name() const {
        return "DetectorEfficiencyCorUser";
}
;

/// Algorithm's version for identification. @see Algorithm::version
int DetectorEfficiencyCorUser::version() const {
        return 1;
}
;

/// Algorithm's category for identification. @see Algorithm::category
const std::string DetectorEfficiencyCorUser::category() const {
        return "CorrectionFunctions\\EfficiencyCorrections;Inelastic";
}

//----------------------------------------------------------------------------------------------

//----------------------------------------------------------------------------------------------
/** Initialize the algorithm's properties.
 */
void DetectorEfficiencyCorUser::init() {
        auto val = boost::make_shared<CompositeValidator>();
        //val->add<WorkspaceUnitValidator>("Energy");
        val->add<WorkspaceUnitValidator>("DeltaE");
        val->add<HistogramValidator>();
        val->add<InstrumentValidator>();
        declareProperty(
                        new WorkspaceProperty<>("InputWorkspace", "", Direction::Input,
                                        val), "The workspace to correct for detector efficiency");
        declareProperty(
                        new WorkspaceProperty<>("OutputWorkspace", "", Direction::Output),
                        "The name of the workspace in which to store the result.");
        auto checkEi = boost::make_shared<BoundedValidator<double> >();
        checkEi->setLower(0.0);
        declareProperty("IncidentEnergy", EMPTY_DBL(), checkEi,
                        "The energy of neutrons leaving the source.");
}

//----------------------------------------------------------------------------------------------
/** Execute the algorithm.
 */
void DetectorEfficiencyCorUser::exec() {

        // get input properties (WSs, Ei)
        retrieveProperties();

        // get Efficiency formula from the IDF
        const std::string effFormula = getValFromInstrumentDef("formula_eff");

        // Calculate Efficiency for E = Ei
        const double eff0 = calculateFormulaValue(effFormula, m_Ei);

        const size_t numberOfChannels = this->m_inputWS->blocksize();
        // Calculate the number of spectra in this workspace
        const int numberOfSpectra = static_cast<int>(this->m_inputWS->size()
                        / numberOfChannels);
        API::Progress prog(this, 0.0, 1.0, numberOfSpectra);
        int64_t numberOfSpectra_i = static_cast<int64_t>(numberOfSpectra); // cast to make openmp happy

        // Loop over the histograms (detector spectra)
        PARALLEL_FOR2(m_outputWS,m_inputWS)
        for (int64_t i = 0; i < numberOfSpectra_i; ++i) {
                PARALLEL_START_INTERUPT_REGION

                //MantidVec& xOut = m_outputWS->dataX(i);
                MantidVec& yOut = m_outputWS->dataY(i);
                MantidVec& eOut = m_outputWS->dataE(i);
                const MantidVec& xIn = m_inputWS->readX(i);
                const MantidVec& yIn = m_inputWS->readY(i);
                const MantidVec& eIn = m_inputWS->readE(i);
                m_outputWS->setX(i, m_inputWS->refX(i));
```

```
                const MantidVec effVec = calculateEfficiency(eff0, effFormula, xIn);
                // run this outside to benefit from parallel for (?)
                applyDetEfficiency(numberOfChannels, yIn, eIn, effVec, yOut, eOut);

                prog.report("Detector Efficiency correction...");

        PARALLEL_END_INTERUPT_REGION
        } //end for i
PARALLEL_CHECK_INTERUPT_REGION

        this->setProperty("OutputWorkspace", this->m_outputWS);

}

/**
 * Apply the detector efficiency to a single spectrum
 * @param numberOfChannels Number of channels in a spectra (nbins - 1)
 * @param yIn spectrum counts
 * @param eIn spectrum errors
 * @param effVec efficiency values (to be divided by the counts)
 * @param yOut corrected spectrum counts
 * @param eOut corrected spectrum errors
 */
void DetectorEfficiencyCorUser::applyDetEfficiency(
                const size_t numberOfChannels, const MantidVec& yIn,
                const MantidVec& eIn, const MantidVec& effVec, MantidVec& yOut,
                MantidVec& eOut) {

        for (unsigned int j = 0; j < numberOfChannels; ++j) {
                //xOut[j] = xIn[j];
                yOut[j] = yIn[j] / effVec[j];
                eOut[j] = eIn[j] / effVec[j];
        }

}
/**
 * Calculate the value of a formula
 * @param formula :: Formula
 * @param energy :: value to use in the formula
 * @return value calculated
 */
double DetectorEfficiencyCorUser::calculateFormulaValue(
                const std::string &formula, double energy) {
        try {
                mu::Parser p;
                p.DefineVar("e", &energy);
                p.SetExpr(formula);
                double eff = p.Eval();
                g_log.debug() << "Formula: " << formula <<  " with: " << energy << "evaluated to: "<< eff  << std::endl;
                return eff;

        } catch (mu::Parser::exception_type &e) {
                throw Kernel::Exception::InstrumentDefinitionError(
                                "Error calculating formula from string. Muparser error message is: "
                                                + e.GetMsg());
        }
}

//MantidVec DetectorEfficiencyCorUser::calculateEfficiency(double eff0,
//              const std::string& formula, const MantidVec& xIn) {
//
//      MantidVec effOut(xIn.size() - 1); // x are bins and have more one value than y
//
//      MantidVec::const_iterator xIn_it = xIn.begin();
//      MantidVec::iterator effOut_it = effOut.begin();
//      for (; effOut_it != effOut.end(); ++xIn_it, ++effOut_it) {
//              double deltaE = std::fabs((*xIn_it + *(xIn_it + 1)) / 2 - m_Ei);
//              double e = m_Ei - deltaE;
//
//              double eff = calculateFormulaValue(formula, e);
//              *effOut_it = eff / eff0;
//      }
//      return effOut;
//}

/**
 * Calculate detector efficiency given a formula, the efficiency at the elastic line,
 * and a vector with energies.
 *  Efficiency = f(Ei-DeltaE) / f(Ei)
 * Hope all compilers supports the NRVO (otherwise will copy the output vector)
 * @param eff0 :: calculated eff0
 * @param formula :: formula to calculate efficiency (parsed from IDF)
 * @param xIn :: Energy bins vector (X axis)
 * @return a vector with the efficiencies
 */
MantidVec DetectorEfficiencyCorUser::calculateEfficiency(double eff0,
                const std::string& formula, const MantidVec& xIn) {

        MantidVec effOut(xIn.size() - 1); // x are bins and have more one value than y

        try {
                double e;
                mu::Parser p;
                p.DefineVar("e", &e);
                p.SetExpr(formula);

                // copied from Jaques Ollivier Code
                bool conditionForEnergy = std::min( std::abs( *std::min_element(xIn.begin(), xIn.end()) ) , m_Ei) < m_Ei;

                MantidVec::const_iterator xIn_it = xIn.begin(); // DeltaE
                MantidVec::iterator effOut_it = effOut.begin();
                for (; effOut_it != effOut.end(); ++xIn_it, ++effOut_it) {
                        if (conditionForEnergy ) {
        // cppcheck cannot see that this is used by reference by muparser
        e =  std::fabs(m_Ei + *xIn_it);
                        }
                        else {
        // cppcheck cannot see that this is used by reference by muparser
        // cppcheck-suppress unreadVariable
                                e =  std::fabs(m_Ei - *xIn_it);
                        }
                        double eff = p.Eval();
                        *effOut_it = eff / eff0;
                }
                return effOut;
```

```
        } catch (mu::Parser::exception_type &e) {
                throw Kernel::Exception::InstrumentDefinitionError(
                                "Error calculating formula from string. Muparser error message is: "
                                                + e.GetMsg());
        }

}
/**
 * Returns the value associated to a parameter name in the IDF
 * @param parameterName :: parameter name in the IDF
 * @return the value associated to the parameter name
 */
std::string DetectorEfficiencyCorUser::getValFromInstrumentDef(
                const std::string& parameterName) {

        const ParameterMap& pmap = m_inputWS->constInstrumentParameters();
        Instrument_const_sptr instrument = m_inputWS->getInstrument();
        Parameter_sptr par = pmap.getRecursive(instrument->getChild(0).get(),
                        parameterName);
        if (par) {
                std::string ret = par->asString();
                g_log.debug() << "Parsed parameter " << parameterName << ": " << ret
                                << "\n";
                return ret;
        } else {
                throw Kernel::Exception::InstrumentDefinitionError(
                                "There is no <" + parameterName
                                                + "> in the instrument definition!");
        }
}

/** Loads and checks the values passed to the algorithm
 *
 *  @throw invalid_argument if there is an incapatible property value so the algorithm can't continue
 */
void DetectorEfficiencyCorUser::retrieveProperties() {

        // Get the workspaces
        m_inputWS = this->getProperty("InputWorkspace");

        m_outputWS = this->getProperty("OutputWorkspace");

        // If input and output workspaces are not the same, create a new workspace for the output
        if (m_outputWS != this->m_inputWS) {
                m_outputWS = API::WorkspaceFactory::Instance().create(m_inputWS);
        }

        // these first three properties are fully checked by validators
        m_Ei = this->getProperty("IncidentEnergy");
        // If we're not given an Ei, see if one has been set.
        if (m_Ei == EMPTY_DBL()) {
                Mantid::Kernel::Property* prop = m_inputWS->run().getProperty("Ei");
                double val;
                if (!prop || !Strings::convert(prop->value(), val)) {
                        throw std::invalid_argument(
                                        "No Ei value has been set or stored within the run information.");
                }
                m_Ei = val;
                g_log.debug() << "Using stored Ei value " << m_Ei << "\n";
        } else {
                g_log.debug() << "Using user input Ei value: " << m_Ei << "\n";
        }
}

}
// namespace Algorithms
}// namespace Mantid
```

SaveILLCosmosAscii	Saves a 2D workspace to a ascii file usable by COSMOS/LAMP	SANS

```
//----------------------------------------------------------------------
// Includes
//----------------------------------------------------------------------
#include "MantidDataHandling/SaveILLCosmosAscii.h"
#include "MantidDataHandling/AsciiPointBase.h"
#include "MantidKernel/ArrayProperty.h"
#include <fstream>

namespace Mantid
{
  namespace DataHandling
  {
    // Register the algorithm into the algorithm factory
    DECLARE_ALGORITHM(SaveILLCosmosAscii)
    using namespace Kernel;
    using namespace API;

    /// virtual method to set the extra properties required for this algorithm
    void SaveILLCosmosAscii::extraProps()
    {
      declareProperty(new ArrayProperty<std::string>("LogList"),"List of logs to write to file.");
      declareProperty("UserContact", "", "Text to be written to the User-local contact field");
      declareProperty("Title", "", "Text to be written to the Title field");
    }

    /** virtual method to add information to the file before the data
     *  @param file :: pointer to output file stream
     */
    void SaveILLCosmosAscii::extraHeaders(std::ofstream & file)
    {
      auto samp = m_ws->run();
      std::string instrument;
      std::string user = getProperty("UserContact");
      std::string title = getProperty("Title");
      std::string subtitle;
      std::string startDT;
      std::string endDT;
      auto tempInst = m_ws->getInstrument();
      if (tempInst)
      {
        instrument = tempInst->getName();
      }

      try
      {
        subtitle = samp.getLogData("run_title")->value();
      }
      catch (Kernel::Exception::NotFoundError &)
      {
        subtitle = "";
      }

      try
      {
        startDT = samp.getLogData("run_start")->value();
      }
      catch (Kernel::Exception::NotFoundError &)
      {
        startDT = "";
      }

      try
      {
        endDT = samp.getLogData("run_end")->value();
      }
      catch (Kernel::Exception::NotFoundError &)
      {
        endDT = "";
      }

      file << "MFT" << std::endl;
      file << "Instrument: "<< instrument << std::endl;
      file << "User-local contact: " << user << std::endl; //add optional property
      file << "Title: " << title << std::endl;
      file << "Subtitle: " << subtitle << std::endl;
      file << "Start date + time: " << startDT << std::endl;
      file << "End date + time: " << endDT << std::endl;

      const std::vector<std::string> logList = getProperty("LogList");
      ///logs
      for (auto log = logList.begin(); log != logList.end(); ++log)
      {
        file << boost::lexical_cast<std::string>(*log) << ": " << boost::lexical_cast<std::string>(samp.getLogData(*log)->value()) <<
std::endl;
      }

      file << "Number of file format: 2" << std::endl;
      file << "Number of data points:" << sep() << m_xlength << std::endl;
      file << std::endl;

      file << sep() << "q" << sep() << "refl" << sep() << "refl_err" << sep() << "q_res" << std::endl;
    }
  } // namespace DataHandling
} // namespace Mantid
```

SetupILLD33Reduction	Set up ILL D33 SANS reduction options.	SANS (ILL: D33)

```
//----------------------------------------------------------------------
// Includes
//----------------------------------------------------------------------
#include "MantidWorkflowAlgorithms/SetupILLD33Reduction.h"
#include "MantidKernel/BoundedValidator.h"
#include "MantidKernel/ListValidator.h"
#include "MantidKernel/RebinParamsValidator.h"
#include "MantidKernel/EnabledWhenProperty.h"
#include "MantidKernel/VisibleWhenProperty.h"
#include "MantidAPI/FileProperty.h"
#include "MantidKernel/ArrayProperty.h"
#include "MantidAPI/AlgorithmProperty.h"
#include "MantidAPI/PropertyManagerDataService.h"
#include "MantidKernel/PropertyManager.h"
#include "Poco/NumberFormatter.h"

namespace Mantid
{
namespace WorkflowAlgorithms
{

// Register the algorithm into the AlgorithmFactory
DECLARE_ALGORITHM(SetupILLD33Reduction)

using namespace Kernel;
using namespace API;
using namespace Geometry;

void SetupILLD33Reduction::init()
{
  // Load options
  std::string load_grp = "Load Options";

  declareProperty("SolidAngleCorrection", true, "If true, the solide angle correction will be applied to the data");
  declareProperty("DetectorTubes", false, "If true, the solid angle correction for tube detectors will be applied");

  // -- Define group --
  setPropertyGroup("SolidAngleCorrection", load_grp);
  setPropertyGroup("DetectorTubes", load_grp);

  // Beam center
  std::string center_grp = "Beam Center";
  std::vector<std::string> centerOptions;
  centerOptions.push_back("None");
  centerOptions.push_back("Value");
  centerOptions.push_back("DirectBeam");
  centerOptions.push_back("Scattering");

  declareProperty("BeamCenterMethod", "None",
      boost::make_shared<StringListValidator>(centerOptions),
      "Method for determining the data beam center");

  //    Option 1: Set beam center by hand
  declareProperty("BeamCenterX", EMPTY_DBL(), "Position of the beam center, in pixel");
  declareProperty("BeamCenterY", EMPTY_DBL(), "Position of the beam center, in pixel");
  setPropertySettings("BeamCenterX",
            new VisibleWhenProperty("BeamCenterMethod", IS_EQUAL_TO, "Value"));
  setPropertySettings("BeamCenterY",
            new VisibleWhenProperty("BeamCenterMethod", IS_EQUAL_TO, "Value"));

  //    Option 2: Find it (expose properties from FindCenterOfMass)
  declareProperty(new API::FileProperty("BeamCenterFile", "", API::FileProperty::OptionalLoad, "_event.nxs"),
      "The name of the input event Nexus file to load");
  setPropertySettings("BeamCenterFile",
            new VisibleWhenProperty("BeamCenterMethod", IS_NOT_EQUAL_TO, "None"));

  auto positiveDouble = boost::make_shared<BoundedValidator<double> >();
  positiveDouble->setLower(0);
  declareProperty("BeamRadius", EMPTY_DBL(),
      "Radius of the beam area used the exclude the beam when calculating "
      "the center of mass of the scattering pattern [pixels]. Default=3.0");
  setPropertySettings("BeamRadius",
            new VisibleWhenProperty("BeamCenterMethod", IS_EQUAL_TO, "Scattering"));

  // -- Define group --
  setPropertyGroup("BeamCenterMethod", center_grp);
  setPropertyGroup("BeamCenterX", center_grp);
  setPropertyGroup("BeamCenterY", center_grp);
  setPropertyGroup("BeamCenterFile", center_grp);
  setPropertyGroup("BeamRadius", center_grp);


  // Normalisation
  std::string norm_grp = "Normalisation";
  std::vector<std::string> incidentBeamNormOptions;
  incidentBeamNormOptions.push_back("None");
  // The data will be normalised to the monitor counts
  incidentBeamNormOptions.push_back("Monitor");
  // The data will be normalised to the total charge only (no beam profile)
  incidentBeamNormOptions.push_back("Timer");
  this->declareProperty("Normalisation", "None",
      boost::make_shared<StringListValidator>(incidentBeamNormOptions),
      "Options for data normalisation");

  setPropertyGroup("Normalisation", norm_grp);

  // Dark current
  declareProperty(new API::FileProperty("DarkCurrentFile", "", API::FileProperty::OptionalLoad, "_event.nxs"),
      "The name of the input event Nexus file to load as dark current.");

  // Sensitivity
  std::string eff_grp = "Sensitivity";
  declareProperty(new API::FileProperty("SensitivityFile", "", API::FileProperty::OptionalLoad, "_event.nxs"),
      "Flood field or sensitivity file.");
  declareProperty("MinEfficiency", EMPTY_DBL(), positiveDouble,
      "Minimum efficiency for a pixel to be considered (default: no minimum).");
  declareProperty("MaxEfficiency", EMPTY_DBL(), positiveDouble,
      "Maximum efficiency for a pixel to be considered (default: no maximum).");
  declareProperty("UseDefaultDC", true, "If true, the dark current subtracted from the sample data will also be subtracted from the flood
```

```
field.");
  declareProperty(new API::FileProperty("SensitivityDarkCurrentFile", "", API::FileProperty::OptionalLoad, "_event.nxs"),
      "The name of the input file to load as dark current.");
  // - sensitivity beam center
  declareProperty("SensitivityBeamCenterMethod", "None",
      boost::make_shared<StringListValidator>(centerOptions),
      "Method for determining the sensitivity data beam center");

  //    Option 1: Set beam center by hand
  declareProperty("SensitivityBeamCenterX", EMPTY_DBL(),
      "Sensitivity beam center location in X [pixels]");
  setPropertySettings("SensitivityBeamCenterX",
            new VisibleWhenProperty("SensitivityBeamCenterMethod", IS_EQUAL_TO, "Value"));

  declareProperty("SensitivityBeamCenterY", EMPTY_DBL(),
      "Sensitivity beam center location in Y [pixels]");
  setPropertySettings("SensitivityBeamCenterY",
            new VisibleWhenProperty("SensitivityBeamCenterMethod", IS_EQUAL_TO, "Value"));

  //    Option 2: Find it (expose properties from FindCenterOfMass)
  declareProperty(new API::FileProperty("SensitivityBeamCenterFile", "",
      API::FileProperty::OptionalLoad, ".xml"),
      "The name of the input data file to load");
  setPropertySettings("SensitivityBeamCenterFile",
            new VisibleWhenProperty("SensitivityBeamCenterMethod", IS_NOT_EQUAL_TO, "None"));

  declareProperty("SensitivityBeamCenterRadius", EMPTY_DBL(),
      "Radius of the beam area used the exclude the beam when calculating "
      "the center of mass of the scattering pattern [pixels]. Default=3.0");
  setPropertySettings("SensitivityBeamCenterRadius",
            new VisibleWhenProperty("BeamCenterMethod", IS_EQUAL_TO, "Scattering"));

  declareProperty("OutputSensitivityWorkspace", "",
      "Name to give the sensitivity workspace");

  // -- Define group --
  setPropertyGroup("SensitivityFile", eff_grp);
  setPropertyGroup("MinEfficiency", eff_grp);
  setPropertyGroup("MaxEfficiency", eff_grp);
  setPropertyGroup("UseDefaultDC", eff_grp);
  setPropertyGroup("SensitivityDarkCurrentFile", eff_grp);
  setPropertyGroup("SensitivityBeamCenterMethod", eff_grp);
  setPropertyGroup("SensitivityBeamCenterX", eff_grp);
  setPropertyGroup("SensitivityBeamCenterY", eff_grp);
  setPropertyGroup("SensitivityBeamCenterFile", eff_grp);
  setPropertyGroup("SensitivityBeamCenterRadius", eff_grp);
  setPropertyGroup("OutputSensitivityWorkspace", eff_grp);

  // Transmission
  std::string trans_grp = "Transmission";
  std::vector<std::string> transOptions;
  transOptions.push_back("Value");
  transOptions.push_back("DirectBeam");
  declareProperty("TransmissionMethod", "Value",
      boost::make_shared<StringListValidator>(transOptions),
      "Transmission determination method");

  // - Transmission value entered by hand
  declareProperty("TransmissionValue", EMPTY_DBL(), positiveDouble,
      "Transmission value.");
  setPropertySettings("TransmissionValue",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "Value"));
  declareProperty("TransmissionError", EMPTY_DBL(), positiveDouble,
      "Transmission error.");
  setPropertySettings("TransmissionError",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "Value"));

  // - Direct beam method transmission calculation
  declareProperty("TransmissionBeamRadius", 3.0,
      "Radius of the beam area used to compute the transmission [pixels]");
  setPropertySettings("TransmissionBeamRadius",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
  declareProperty(new API::FileProperty("TransmissionSampleDataFile", "",
      API::FileProperty::OptionalLoad, ".xml"),
      "Sample data file for transmission calculation");
  setPropertySettings("TransmissionSampleDataFile",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
  declareProperty(new API::FileProperty("TransmissionEmptyDataFile", "",
      API::FileProperty::OptionalLoad, ".xml"),
      "Empty data file for transmission calculation");
  setPropertySettings("TransmissionEmptyDataFile",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));

  // - transmission beam center
  declareProperty("TransmissionBeamCenterMethod", "None",
      boost::make_shared<StringListValidator>(centerOptions),
      "Method for determining the transmission data beam center");
  setPropertySettings("TransmissionBeamCenterMethod",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));

  //    Option 1: Set beam center by hand
  declareProperty("TransmissionBeamCenterX", EMPTY_DBL(),
      "Transmission beam center location in X [pixels]");
  setPropertySettings("TransmissionBeamCenterX",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
  declareProperty("TransmissionBeamCenterY", EMPTY_DBL(),
      "Transmission beam center location in Y [pixels]");
  setPropertySettings("TransmissionBeamCenterY",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));

  //    Option 2: Find it (expose properties from FindCenterOfMass)
  declareProperty(new API::FileProperty("TransmissionBeamCenterFile", "",
      API::FileProperty::OptionalLoad, ".xml"),
      "The name of the input data file to load");
  setPropertySettings("TransmissionBeamCenterFile",
            new VisibleWhenProperty("TransmissionMethod", IS_EQUAL_TO, "DirectBeam"));

  declareProperty(new API::FileProperty("TransmissionDarkCurrentFile", "", API::FileProperty::OptionalLoad, ".xml"),
      "The name of the input data file to load as transmission dark current.");
  setPropertySettings("TransmissionDarkCurrentFile",
            new VisibleWhenProperty("TransmissionMethod", IS_NOT_EQUAL_TO, "Value"));

  declareProperty("TransmissionUseSampleDC", true,
      "If true, the sample dark current will be used IF a dark current file is"
      "not set.");
```

```
setPropertySettings("TransmissionUseSampleDC",
          new VisibleWhenProperty("TransmissionMethod", IS_NOT_EQUAL_TO, "Value"));

declareProperty("ThetaDependentTransmission", true,
    "If true, a theta-dependent transmission correction will be applied.");

// -- Define group --
setPropertyGroup("TransmissionMethod", trans_grp);
setPropertyGroup("TransmissionValue", trans_grp);
setPropertyGroup("TransmissionError", trans_grp);
setPropertyGroup("TransmissionBeamRadius", trans_grp);
setPropertyGroup("TransmissionSampleDataFile", trans_grp);
setPropertyGroup("TransmissionEmptyDataFile", trans_grp);
setPropertyGroup("TransmissionBeamCenterMethod", trans_grp);
setPropertyGroup("TransmissionBeamCenterX", trans_grp);
setPropertyGroup("TransmissionBeamCenterY", trans_grp);
setPropertyGroup("TransmissionBeamCenterFile", trans_grp);

setPropertyGroup("TransmissionDarkCurrentFile", trans_grp);
setPropertyGroup("TransmissionUseSampleDC", trans_grp);
setPropertyGroup("ThetaDependentTransmission", trans_grp);

// Background options
std::string bck_grp = "Background";
declareProperty("BackgroundFiles", "", "Background data files");
declareProperty("BckTransmissionMethod", "Value",
    boost::make_shared<StringListValidator>(transOptions),
    "Transmission determination method");

// - Transmission value entered by hand
declareProperty("BckTransmissionValue", EMPTY_DBL(), positiveDouble,
    "Transmission value.");
setPropertySettings("BckTransmissionValue",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "Value"));

declareProperty("BckTransmissionError", EMPTY_DBL(), positiveDouble,
    "Transmission error.");
setPropertySettings("BckTransmissionError",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "Value"));

// - Direct beam method transmission calculation
declareProperty("BckTransmissionBeamRadius", 3.0,
    "Radius of the beam area used to compute the transmission [pixels]");
setPropertySettings("BckTransmissionBeamRadius",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
declareProperty(new API::FileProperty("BckTransmissionSampleDataFile", "",
    API::FileProperty::OptionalLoad, ".xml"),
    "Sample data file for transmission calculation");
setPropertySettings("BckTransmissionSampleDataFile",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
declareProperty(new API::FileProperty("BckTransmissionEmptyDataFile", "",
    API::FileProperty::OptionalLoad, ".xml"),
    "Empty data file for transmission calculation");
setPropertySettings("BckTransmissionEmptyDataFile",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));

// - transmission beam center
declareProperty("BckTransmissionBeamCenterMethod", "None",
    boost::make_shared<StringListValidator>(centerOptions),
    "Method for determining the transmission data beam center");
setPropertySettings("BckTransmissionBeamCenterMethod",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
//    Option 1: Set beam center by hand
declareProperty("BckTransmissionBeamCenterX", EMPTY_DBL(),
    "Transmission beam center location in X [pixels]");
setPropertySettings("BckTransmissionBeamCenterX",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
declareProperty("BckTransmissionBeamCenterY", EMPTY_DBL(),
    "Transmission beam center location in Y [pixels]");
//    Option 2: Find it (expose properties from FindCenterOfMass)
setPropertySettings("BckTransmissionBeamCenterY",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));
declareProperty(new API::FileProperty("BckTransmissionBeamCenterFile", "",
    API::FileProperty::OptionalLoad, ".xml"),
    "The name of the input data file to load");
setPropertySettings("BckTransmissionBeamCenterFile",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "DirectBeam"));

declareProperty(new API::FileProperty("BckTransmissionDarkCurrentFile", "", API::FileProperty::OptionalLoad, ".xml"),
    "The name of the input data file to load as background transmission dark current.");
setPropertySettings("BckTransmissionDarkCurrentFile",
          new VisibleWhenProperty("BckTransmissionMethod", IS_EQUAL_TO, "BeamSpreader"));

declareProperty("BckThetaDependentTransmission", true,
    "If true, a theta-dependent transmission correction will be applied.");

setPropertyGroup("BackgroundFiles", bck_grp);
setPropertyGroup("BckTransmissionMethod", bck_grp);
setPropertyGroup("BckTransmissionValue", bck_grp);
setPropertyGroup("BckTransmissionError", bck_grp);
setPropertyGroup("BckTransmissionBeamRadius", bck_grp);
setPropertyGroup("BckTransmissionSampleDataFile", bck_grp);
setPropertyGroup("BckTransmissionEmptyDataFile", bck_grp);
setPropertyGroup("BckTransmissionBeamCenterMethod", bck_grp);
setPropertyGroup("BckTransmissionBeamCenterX", bck_grp);
setPropertyGroup("BckTransmissionBeamCenterY", bck_grp);
setPropertyGroup("BckTransmissionBeamCenterFile", bck_grp);
setPropertyGroup("BckTransmissionDarkCurrentFile", bck_grp);
setPropertyGroup("BckThetaDependentTransmission", bck_grp);

// Geometry correction
declareProperty("SampleThickness", EMPTY_DBL(), "Sample thickness [cm]");

// Masking
std::string mask_grp = "Mask";
declareProperty(new ArrayProperty<int>("MaskedDetectorList"),
    "List of detector IDs to be masked");
declareProperty(new ArrayProperty<int>("MaskedEdges"),
    "Number of pixels to mask on the edges: X-low, X-high, Y-low, Y-high");
std::vector<std::string> maskOptions;
maskOptions.push_back("None");
maskOptions.push_back("Front");
maskOptions.push_back("Back");
declareProperty("MaskedSide", "None",
    boost::make_shared<StringListValidator>(maskOptions),
```

```
        "Mask one side of the detector");

    setPropertyGroup("MaskedDetectorList", mask_grp);
    setPropertyGroup("MaskedEdges", mask_grp);
    setPropertyGroup("MaskedSide", mask_grp);

    // Absolute scale
    std::string abs_scale_grp = "Absolute Scale";
    std::vector<std::string> scaleOptions;
    scaleOptions.push_back("None");
    scaleOptions.push_back("Value");
    scaleOptions.push_back("ReferenceData");
    declareProperty("AbsoluteScaleMethod", "None",
        boost::make_shared<StringListValidator>(scaleOptions),
        "Absolute scale correction method");
    declareProperty("AbsoluteScalingFactor", 1.0, "Absolute scaling factor");
    setPropertySettings("AbsoluteScalingFactor",
        new VisibleWhenProperty("AbsoluteScaleMethod", IS_EQUAL_TO, "Value"));

    declareProperty(new API::FileProperty("AbsoluteScalingReferenceFilename", "",
        API::FileProperty::OptionalLoad, ".xml"));
    setPropertySettings("AbsoluteScalingReferenceFilename",
             new VisibleWhenProperty("AbsoluteScaleMethod", IS_EQUAL_TO, "ReferenceData"));
    declareProperty("AbsoluteScalingBeamDiameter", 0.0,
        "Beamstop diameter for computing the absolute scale factor [mm]. "
        "Read from file if not supplied.");
    setPropertySettings("AbsoluteScalingBeamDiameter",
        new VisibleWhenProperty("AbsoluteScaleMethod", IS_EQUAL_TO, "ReferenceData"));
    declareProperty("AbsoluteScalingAttenuatorTrans", 1.0,
        "Attenuator transmission value for computing the absolute scale factor");
    setPropertySettings("AbsoluteScalingAttenuatorTrans",
        new VisibleWhenProperty("AbsoluteScaleMethod", IS_EQUAL_TO, "ReferenceData"));
    declareProperty("AbsoluteScalingApplySensitivity", false,
        "Apply sensitivity correction to the reference data "
        "when computing the absolute scale factor");
    setPropertySettings("AbsoluteScalingApplySensitivity",
        new VisibleWhenProperty("AbsoluteScaleMethod", IS_EQUAL_TO, "ReferenceData"));

    setPropertyGroup("AbsoluteScaleMethod", abs_scale_grp);
    setPropertyGroup("AbsoluteScalingFactor", abs_scale_grp);
    setPropertyGroup("AbsoluteScalingReferenceFilename", abs_scale_grp);
    setPropertyGroup("AbsoluteScalingBeamDiameter", abs_scale_grp);
    setPropertyGroup("AbsoluteScalingAttenuatorTrans", abs_scale_grp);
    setPropertyGroup("AbsoluteScalingApplySensitivity", abs_scale_grp);

    // I(Q) calculation
    std::string iq1d_grp = "I(q) Calculation";
    declareProperty("DoAzimuthalAverage", true);
    auto positiveInt = boost::make_shared<BoundedValidator<int> >();
    positiveInt->setLower(0);
    declareProperty("IQNumberOfBins", 100, positiveInt,
                    "Number of I(q) bins when binning is not specified");
    declareProperty("IQLogBinning", false,
                    "I(q) log binning when binning is not specified");
    declareProperty("ComputeResolution", false,
                    "If true the Q resolution will be computed");

    declareProperty("Do2DReduction", true);
    declareProperty("IQ2DNumberOfBins", 100, positiveInt,
                    "Number of I(qx,qy) bins.");

    // -- Define group --
    setPropertyGroup("DoAzimuthalAverage", iq1d_grp);
    setPropertyGroup("IQNumberOfBins", iq1d_grp);
    setPropertyGroup("IQLogBinning", iq1d_grp);
    setPropertyGroup("ComputeResolution", iq1d_grp);
    setPropertyGroup("Do2DReduction", iq1d_grp);
    setPropertyGroup("IQ2DNumberOfBins", iq1d_grp);

    // Outputs
    declareProperty("ProcessInfo","", "Additional process information");
    declareProperty("OutputDirectory", "", "Directory to put the output files in");
    declareProperty("OutputMessage","",Direction::Output);
    declareProperty("ReductionProperties","__sans_reduction_properties", Direction::Input);
}

void SetupILLD33Reduction::exec()
{
    // Reduction property manager
    const std::string reductionManagerName = getProperty("ReductionProperties");
    if (reductionManagerName.size()==0)
    {
      g_log.error() << "ERROR: Reduction Property Manager name is empty" << std::endl;
      return;
    }
    boost::shared_ptr<PropertyManager> reductionManager = boost::make_shared<PropertyManager>();
    PropertyManagerDataService::Instance().addOrReplace(reductionManagerName, reductionManager);

    // Store name of the instrument
    reductionManager->declareProperty(new PropertyWithValue<std::string>("InstrumentName", "D33") );

    // Store additional (and optional) process information
    const std::string processInfo = getProperty("ProcessInfo");
    reductionManager->declareProperty(new PropertyWithValue<std::string>("ProcessInfo", processInfo));

    // Store the output directory
    const std::string outputDirectory = getProperty("OutputDirectory");
    reductionManager->declareProperty(new PropertyWithValue<std::string>("OutputDirectory", outputDirectory));

    // Store normalization algorithm
    const std::string normalization = getProperty("Normalisation");

    if (!boost::contains(normalization, "None")) {
            // If we normalize to monitor, force the loading of monitor data
            IAlgorithm_sptr normAlg = createChildAlgorithm("HFIRSANSNormalise");
            normAlg->setProperty("NormalisationType", normalization);
            //normAlg->setPropertyValue("ReductionProperties", reductionManagerName);
            AlgorithmProperty *algProp = new AlgorithmProperty("NormaliseAlgorithm");
            algProp->setValue(normAlg->toString());
            reductionManager->declareProperty(algProp);
    }

    // Load algorithm
    //IAlgorithm_sptr loadAlg = createChildAlgorithm("EQSANSLoad");
    // TODO : It looks like properties need cleanup
```

```
IAlgorithm_sptr loadAlg = createChildAlgorithm("LoadILLSANS");

AlgorithmProperty *algProp = new AlgorithmProperty("LoadAlgorithm");
algProp->setValue(loadAlg->toString());
reductionManager->declareProperty(algProp);

// Store dark current algorithm
const std::string darkCurrentFile = getPropertyValue("DarkCurrentFile");
if (darkCurrentFile.size() > 0)
{
  IAlgorithm_sptr darkAlg = createChildAlgorithm("EQSANSDarkCurrentSubtraction");
  darkAlg->setProperty("Filename", darkCurrentFile);
  darkAlg->setProperty("OutputDarkCurrentWorkspace", "");
  darkAlg->setPropertyValue("ReductionProperties", reductionManagerName);
  algProp = new AlgorithmProperty("DarkCurrentAlgorithm");
  algProp->setValue(darkAlg->toString());
  reductionManager->declareProperty(algProp);
}

// Store default dark current algorithm
IAlgorithm_sptr darkDefaultAlg = createChildAlgorithm("EQSANSDarkCurrentSubtraction");
darkDefaultAlg->setProperty("OutputDarkCurrentWorkspace", "");
darkDefaultAlg->setPropertyValue("ReductionProperties", reductionManagerName);
algProp = new AlgorithmProperty("DefaultDarkCurrentAlgorithm");
algProp->setValue(darkDefaultAlg->toString());
reductionManager->declareProperty(algProp);

// Solid angle correction
const bool solidAngleCorrection = getProperty("SolidAngleCorrection");
if (solidAngleCorrection)
{
  const bool detectorTubes = getProperty("DetectorTubes");
  IAlgorithm_sptr solidAlg = createChildAlgorithm("SANSSolidAngleCorrection");
  solidAlg->setProperty("DetectorTubes", detectorTubes);
  algProp = new AlgorithmProperty("SANSSolidAngleCorrection");
  algProp->setValue(solidAlg->toString());
  reductionManager->declareProperty(algProp);
}

// Beam center
const double beamCenterX = getProperty("BeamCenterX");
const double beamCenterY = getProperty("BeamCenterY");
const std::string centerMethod = getPropertyValue("BeamCenterMethod");

// Beam center option for transmission data
if (boost::iequals(centerMethod, "Value"))
{
  if(!isEmpty(beamCenterX) && !isEmpty(beamCenterY))
  {
    reductionManager->declareProperty(new PropertyWithValue<double>("LatestBeamCenterX", beamCenterX) );
    reductionManager->declareProperty(new PropertyWithValue<double>("LatestBeamCenterY", beamCenterY) );
  }
}
else if (!boost::iequals(centerMethod, "None"))
{
  bool useDirectBeamMethod = true;
  if (!boost::iequals(centerMethod, "DirectBeam")) useDirectBeamMethod = false;
  const std::string beamCenterFile = getProperty("BeamCenterFile");
  if (beamCenterFile.size()>0)
  {
    const double beamRadius = getProperty("BeamRadius");

    IAlgorithm_sptr ctrAlg = createChildAlgorithm("SANSBeamFinder");
    ctrAlg->setProperty("Filename", beamCenterFile);
    ctrAlg->setProperty("UseDirectBeamMethod", useDirectBeamMethod);
    if (!isEmpty(beamRadius)) ctrAlg->setProperty("BeamRadius", beamRadius);
    ctrAlg->setPropertyValue("ReductionProperties", reductionManagerName);

    AlgorithmProperty *algProp = new AlgorithmProperty("SANSBeamFinderAlgorithm");
    algProp->setValue(ctrAlg->toString());
    reductionManager->declareProperty(algProp);
  } else {
    g_log.error() << "ERROR: Beam center determination was required"
        " but no file was provided" << std::endl;
  }
}

// Sensitivity correction, transmission and background
setupSensitivity(reductionManager);
setupTransmission(reductionManager);
setupBackground(reductionManager);

// Geometry correction
const double thickness = getProperty("SampleThickness");
if (!isEmpty(thickness))
{
  IAlgorithm_sptr thickAlg = createChildAlgorithm("NormaliseByThickness");
  thickAlg->setProperty("SampleThickness", thickness);

  algProp = new AlgorithmProperty("GeometryAlgorithm");
  algProp->setValue(thickAlg->toString());
  reductionManager->declareProperty(algProp);
}

// Mask
const std::string maskDetList = getPropertyValue("MaskedDetectorList");
const std::string maskEdges = getPropertyValue("MaskedEdges");
const std::string maskSide = getProperty("MaskedSide");

IAlgorithm_sptr maskAlg = createChildAlgorithm("SANSMask");
// The following is broken, try PropertyValue
maskAlg->setPropertyValue("Facility", "SNS");
maskAlg->setPropertyValue("MaskedDetectorList", maskDetList);
maskAlg->setPropertyValue("MaskedEdges", maskEdges);
maskAlg->setProperty("MaskedSide", maskSide);
algProp = new AlgorithmProperty("MaskAlgorithm");
algProp->setValue(maskAlg->toString());
reductionManager->declareProperty(algProp);

// Absolute scaling
const std::string absScaleMethod = getProperty("AbsoluteScaleMethod");
if (boost::iequals(absScaleMethod, "Value"))
{
  const double absScaleFactor = getProperty("AbsoluteScalingFactor");
```

```
    IAlgorithm_sptr absAlg = createChildAlgorithm("SANSAbsoluteScale");
    absAlg->setProperty("Method", absScaleMethod);
    absAlg->setProperty("ScalingFactor", absScaleFactor);
    absAlg->setPropertyValue("ReductionProperties", reductionManagerName);
    algProp = new AlgorithmProperty("AbsoluteScaleAlgorithm");
    algProp->setValue(absAlg->toString());
    reductionManager->declareProperty(algProp);
  }
  else if (boost::iequals(absScaleMethod, "ReferenceData"))
  {
    const std::string absRefFile = getPropertyValue("AbsoluteScalingReferenceFilename");
    const double beamDiam = getProperty("AbsoluteScalingBeamDiameter");
    const double attTrans = getProperty("AbsoluteScalingAttenuatorTrans");
    const bool applySensitivity = getProperty("AbsoluteScalingApplySensitivity");

    IAlgorithm_sptr absAlg = createChildAlgorithm("SANSAbsoluteScale");
    absAlg->setProperty("Method", absScaleMethod);
    absAlg->setProperty("ReferenceDataFilename", absRefFile);
    absAlg->setProperty("BeamstopDiameter", beamDiam);
    absAlg->setProperty("AttenuatorTransmission", attTrans);
    absAlg->setProperty("ApplySensitivity", applySensitivity);
    absAlg->setPropertyValue("ReductionProperties", reductionManagerName);
    algProp = new AlgorithmProperty("AbsoluteScaleAlgorithm");
    algProp->setValue(absAlg->toString());
    reductionManager->declareProperty(algProp);
  }

  // Azimuthal averaging
  const bool doAveraging = getProperty("DoAzimuthalAverage");
  if (doAveraging)
  {
    const std::string nBins = getPropertyValue("IQNumberOfBins");
    const bool logBinning = getProperty("IQLogBinning");
    const bool computeResolution = getProperty("ComputeResolution");

    IAlgorithm_sptr iqAlg = createChildAlgorithm("SANSAzimuthalAverage1D");
    iqAlg->setPropertyValue("NumberOfBins", nBins);
    iqAlg->setProperty("LogBinning", logBinning);
    iqAlg->setProperty("ComputeResolution", computeResolution);
    iqAlg->setPropertyValue("ReductionProperties", reductionManagerName);

    algProp = new AlgorithmProperty("IQAlgorithm");
    algProp->setValue(iqAlg->toString());
    reductionManager->declareProperty(algProp);
  }

  // 2D reduction
  const bool do2DReduction = getProperty("Do2DReduction");
  if (do2DReduction)
  {
    const std::string n_bins = getPropertyValue("IQ2DNumberOfBins");
    IAlgorithm_sptr iqAlg = createChildAlgorithm("EQSANSQ2D");
    iqAlg->setPropertyValue("NumberOfBins", n_bins);
    algProp = new AlgorithmProperty("IQXYAlgorithm");
    algProp->setValue(iqAlg->toString());
    reductionManager->declareProperty(algProp);
  }
  setPropertyValue("OutputMessage", "EQSANS reduction options set");
}

void SetupILLD33Reduction::setupSensitivity(boost::shared_ptr<PropertyManager> reductionManager)
{
  const std::string reductionManagerName = getProperty("ReductionProperties");

  const std::string sensitivityFile = getPropertyValue("SensitivityFile");
  if (sensitivityFile.size() > 0)
  {
    const bool useSampleDC = getProperty("UseDefaultDC");
    const std::string sensitivityDarkCurrentFile = getPropertyValue("SensitivityDarkCurrentFile");
    const std::string outputSensitivityWS = getPropertyValue("OutputSensitivityWorkspace");
    const double minEff = getProperty("MinEfficiency");
    const double maxEff = getProperty("MaxEfficiency");
    const double sensitivityBeamCenterX = getProperty("SensitivityBeamCenterX");
    const double sensitivityBeamCenterY = getProperty("SensitivityBeamCenterY");

    IAlgorithm_sptr effAlg = createChildAlgorithm("SANSSensitivityCorrection");
    effAlg->setProperty("Filename", sensitivityFile);
    effAlg->setProperty("UseSampleDC", useSampleDC);
    effAlg->setProperty("DarkCurrentFile", sensitivityDarkCurrentFile);
    effAlg->setProperty("MinEfficiency", minEff);
    effAlg->setProperty("MaxEfficiency", maxEff);

    // Beam center option for sensitivity data
    const std::string centerMethod = getPropertyValue("SensitivityBeamCenterMethod");
    if (boost::iequals(centerMethod, "Value"))
    {
      if (!isEmpty(sensitivityBeamCenterX) &&
          !isEmpty(sensitivityBeamCenterY))
      {
        effAlg->setProperty("BeamCenterX", sensitivityBeamCenterX);
        effAlg->setProperty("BeamCenterY", sensitivityBeamCenterY);
      }
    }
    else if (boost::iequals(centerMethod, "DirectBeam") ||
        boost::iequals(centerMethod, "Scattering"))
    {
      const std::string beamCenterFile = getProperty("SensitivityBeamCenterFile");
      const double sensitivityBeamRadius = getProperty("SensitivityBeamCenterRadius");
      bool useDirectBeam = boost::iequals(centerMethod, "DirectBeam");
      if (beamCenterFile.size()>0)
       {
        IAlgorithm_sptr ctrAlg = createChildAlgorithm("SANSBeamFinder");
        ctrAlg->setProperty("Filename", beamCenterFile);
        ctrAlg->setProperty("UseDirectBeamMethod", useDirectBeam);
        ctrAlg->setProperty("PersistentCorrection", false);
        if (useDirectBeam && !isEmpty(sensitivityBeamRadius))
          ctrAlg->setProperty("BeamRadius", sensitivityBeamRadius);
        ctrAlg->setPropertyValue("ReductionProperties", reductionManagerName);

        AlgorithmProperty *algProp = new AlgorithmProperty("SensitivityBeamCenterAlgorithm");
        algProp->setValue(ctrAlg->toString());
        reductionManager->declareProperty(algProp);
       } else {
        g_log.error() << "ERROR: Sensitivity beam center determination was required"
            " but no file was provided" << std::endl;
```

```
      }
    }

    effAlg->setPropertyValue("OutputSensitivityWorkspace", outputSensitivityWS);
    effAlg->setPropertyValue("ReductionProperties", reductionManagerName);

    AlgorithmProperty *algProp = new AlgorithmProperty("SensitivityAlgorithm");
    algProp->setValue(effAlg->toString());
    reductionManager->declareProperty(algProp);
  }
}
void SetupILLD33Reduction::setupTransmission(boost::shared_ptr<PropertyManager> reductionManager)
{
  const std::string reductionManagerName = getProperty("ReductionProperties");
  // Transmission options
  const bool thetaDependentTrans = getProperty("ThetaDependentTransmission");
  const std::string transMethod = getProperty("TransmissionMethod");
  const std::string darkCurrent = getPropertyValue("TransmissionDarkCurrentFile");
  const bool useSampleDC = getProperty("TransmissionUseSampleDC");

  // Transmission is entered by hand
  if (boost::iequals(transMethod, "Value"))
  {
    const double transValue = getProperty("TransmissionValue");
    const double transError = getProperty("TransmissionError");
    if (!isEmpty(transValue) && !isEmpty(transError))
    {
      IAlgorithm_sptr transAlg = createChildAlgorithm("ApplyTransmissionCorrection");
      transAlg->setProperty("TransmissionValue", transValue);
      transAlg->setProperty("TransmissionError", transError);
      transAlg->setProperty("ThetaDependent", thetaDependentTrans);

      AlgorithmProperty *algProp = new AlgorithmProperty("TransmissionAlgorithm");
      algProp->setValue(transAlg->toString());
      reductionManager->declareProperty(algProp);
    } else {
      g_log.information("SetupILLD33Reduction [TransmissionAlgorithm]:"
          "expected transmission/error values and got empty values");
    }
  }
  // Direct beam method for transmission determination
  else if (boost::iequals(transMethod, "DirectBeam"))
  {
    const std::string sampleFilename = getPropertyValue("TransmissionSampleDataFile");
    const std::string emptyFilename = getPropertyValue("TransmissionEmptyDataFile");
    const double beamRadius = getProperty("TransmissionBeamRadius");
    const double beamX = getProperty("TransmissionBeamCenterX");
    const double beamY = getProperty("TransmissionBeamCenterY");
    const std::string centerMethod = getPropertyValue("TransmissionBeamCenterMethod");

    IAlgorithm_sptr transAlg = createChildAlgorithm("SANSDirectBeamTransmission");
     transAlg->setProperty("SampleDataFilename", sampleFilename);
    transAlg->setProperty("EmptyDataFilename", emptyFilename);
    transAlg->setProperty("BeamRadius", beamRadius);
    transAlg->setProperty("DarkCurrentFilename", darkCurrent);
    transAlg->setProperty("UseSampleDarkCurrent", useSampleDC);

    // Beam center option for transmission data
    if (boost::iequals(centerMethod, "Value") && !isEmpty(beamX) && !isEmpty(beamY))
    {
      transAlg->setProperty("BeamCenterX", beamX);
      transAlg->setProperty("BeamCenterY", beamY);
    }
    else if (boost::iequals(centerMethod, "DirectBeam"))
    {
      const std::string beamCenterFile = getProperty("TransmissionBeamCenterFile");
      if (beamCenterFile.size()>0)
       {
        IAlgorithm_sptr ctrAlg = createChildAlgorithm("SANSBeamFinder");
        ctrAlg->setProperty("Filename", beamCenterFile);
        ctrAlg->setProperty("UseDirectBeamMethod", true);
        ctrAlg->setProperty("PersistentCorrection", false);
        ctrAlg->setPropertyValue("ReductionProperties", reductionManagerName);

        AlgorithmProperty *algProp = new AlgorithmProperty("TransmissionBeamCenterAlgorithm");
        algProp->setValue(ctrAlg->toString());
        reductionManager->declareProperty(algProp);
       } else {
        g_log.error() << "ERROR: Transmission beam center determination was required"
            " but no file was provided" << std::endl;
       }
    }
    transAlg->setProperty("ThetaDependent", thetaDependentTrans);
    AlgorithmProperty *algProp = new AlgorithmProperty("TransmissionAlgorithm");
    algProp->setValue(transAlg->toString());
    reductionManager->declareProperty(algProp);
  }
}

void SetupILLD33Reduction::setupBackground(boost::shared_ptr<PropertyManager> reductionManager)
{
  const std::string reductionManagerName = getProperty("ReductionProperties");
  // Background
  const std::string backgroundFile = getPropertyValue("BackgroundFiles");
  if (backgroundFile.size() > 0)
    reductionManager->declareProperty(new PropertyWithValue<std::string>("BackgroundFiles", backgroundFile) );
  else
    return;

  const std::string darkCurrent = getPropertyValue("BckTransmissionDarkCurrentFile");
  const bool bckThetaDependentTrans = getProperty("BckThetaDependentTransmission");
  const std::string bckTransMethod = getProperty("BckTransmissionMethod");
  if (boost::iequals(bckTransMethod, "Value"))
  {
    const double transValue = getProperty("BckTransmissionValue");
    const double transError = getProperty("BckTransmissionError");
    if (!isEmpty(transValue) && !isEmpty(transError))
    {
      IAlgorithm_sptr transAlg = createChildAlgorithm("ApplyTransmissionCorrection");
      transAlg->setProperty("TransmissionValue", transValue);
      transAlg->setProperty("TransmissionError", transError);
      transAlg->setProperty("ThetaDependent", bckThetaDependentTrans);

      AlgorithmProperty *algProp = new AlgorithmProperty("BckTransmissionAlgorithm");
      algProp->setValue(transAlg->toString());
```

```
      reductionManager->declareProperty(algProp);
    } else {
      g_log.information("SetupILLD33Reduction [BckTransmissionAlgorithm]: "
          "expected transmission/error values and got empty values");
    }
  }
  else if (boost::iequals(bckTransMethod, "DirectBeam"))
  {
    const std::string sampleFilename = getPropertyValue("BckTransmissionSampleDataFile");
    const std::string emptyFilename = getPropertyValue("BckTransmissionEmptyDataFile");
    const double beamRadius = getProperty("BckTransmissionBeamRadius");
    const double beamX = getProperty("BckTransmissionBeamCenterX");
    const double beamY = getProperty("BckTransmissionBeamCenterY");
    const bool thetaDependentTrans = getProperty("BckThetaDependentTransmission");
    const bool useSampleDC = getProperty("TransmissionUseSampleDC");

    IAlgorithm_sptr transAlg = createChildAlgorithm("SANSDirectBeamTransmission");
    transAlg->setProperty("SampleDataFilename", sampleFilename);
    transAlg->setProperty("EmptyDataFilename", emptyFilename);
    transAlg->setProperty("BeamRadius", beamRadius);
    transAlg->setProperty("DarkCurrentFilename", darkCurrent);
    transAlg->setProperty("UseSampleDarkCurrent", useSampleDC);

    // Beam center option for transmission data
    const std::string centerMethod = getPropertyValue("BckTransmissionBeamCenterMethod");
    if (boost::iequals(centerMethod, "Value") && !isEmpty(beamX) && !isEmpty(beamY))
    {
      transAlg->setProperty("BeamCenterX", beamX);
      transAlg->setProperty("BeamCenterY", beamY);
    }
    else if (boost::iequals(centerMethod, "DirectBeam"))
    {
      const std::string beamCenterFile = getProperty("BckTransmissionBeamCenterFile");
      if (beamCenterFile.size()>0)
       {
        IAlgorithm_sptr ctrAlg = createChildAlgorithm("SANSBeamFinder");
        ctrAlg->setProperty("Filename", beamCenterFile);
        ctrAlg->setProperty("UseDirectBeamMethod", true);
        ctrAlg->setProperty("PersistentCorrection", false);
        ctrAlg->setPropertyValue("ReductionProperties", reductionManagerName);

        AlgorithmProperty *algProp = new AlgorithmProperty("BckTransmissionBeamCenterAlgorithm");
        algProp->setValue(ctrAlg->toString());
        reductionManager->declareProperty(algProp);
       } else {
        g_log.error() << "ERROR: Beam center determination was required"
            " but no file was provided" << std::endl;
       }
    }
    transAlg->setProperty("DarkCurrentFilename", darkCurrent);
    transAlg->setProperty("ThetaDependent", thetaDependentTrans);
    AlgorithmProperty *algProp = new AlgorithmProperty("BckTransmissionAlgorithm");
    algProp->setValue(transAlg->toString());
    reductionManager->declareProperty(algProp);
  }

}
} // namespace WorkflowAlgorithms
} // namespace Mantid
```

AllToMantid: communicator

```
'''
Created on Oct 17, 2013

@author: leal
'''

import Queue
import threading
import subprocess
import time
import os


class Communicate(object):
    '''
    Class to communicate asynchronously with a Linux process.
    '''


    def __init__(self, executable, prompt, exitCommand = None):
        """

        @param executable:
        @param prompt: Any string output when program has finished to start
        @param exitCommand: if there is a exit command
        """


        self._executable = executable
        self._prompt = prompt
        self._exitCommand = exitCommand

        self._launch()

        self._outQueue = Queue.Queue()
        self._errQueue = Queue.Queue()

        self._startThreads()
        self._startExecutable()

    def _startThreads(self):
        self.outThread = threading.Thread(target=self._enqueueOutput, args=(self._subproc.stdout, self._outQueue))
        self.errThread = threading.Thread(target=self._enqueueOutput, args=(self._subproc.stderr, self._errQueue))

        self.outThread.daemon = True
        self.errThread.daemon = True

        self.outThread.start()
        self.errThread.start()

    def _startExecutable(self):
        print self._executable, 'is starting...'
        output = self._getOutput(self._outQueue)
        while output.find(self._prompt) < 0:
            time.sleep(0.2)
            output = self._getOutput(self._outQueue)
        print self._executable, 'is starting... Done!'

        errors = self._getOutput(self._errQueue)
        print 'Errors while starting:'
        print errors


    def _enqueueOutput(self, out, queue):
        for line in iter(out.readline, b''):
            queue.put(line)
        out.close()

    def _getOutput(self, outQueue):
        outStr = ''
        try:
            while True:  # Adds output from the Queue until it is empty
                outStr += outQueue.get_nowait()
        except Queue.Empty:
            return outStr

    def send(self,command):
        self._relaunchIfItIsNotRunning()
        self._subproc.stdin.write(command + os.linesep)
        self._subproc.stdin.flush()

    def receiveOutput(self):
        output = self._getOutput(self._outQueue)
        return output

    def receiveErrors(self):
        errors = self._getOutput(self._errQueue)
        return errors

    def communicate(self,command,waitTimeForTheCommandToGiveOutput=0.2):
        print 'Executing:', command
        self.send(command)
        time.sleep(waitTimeForTheCommandToGiveOutput)
        return [self.receiveOutput(),self.receiveErrors()]

    def exit(self):
        """
        Try to send the exit command to subprocess running.
        Kill it if it is still running
        """
        if self._isSubProcessRunning() and self._exitCommand is not None:
            self._subproc.stdin.write(self._exitCommand)
            self._subproc.stdin.write(os.linesep)
            self._subproc.stdin.flush()
            time.sleep(0.5)

        if self._isSubProcessRunning() :
            self._subproc.kill()
        time.sleep(0.1)
        print 'Done!'
```

```
    def _launch(self):
        self._subproc = subprocess.Popen(self._executable, stdin=subprocess.PIPE, stdout=subprocess.PIPE,
                                         stderr=subprocess.PIPE, shell=True, universal_newlines=True)
    def _relaunchIfItIsNotRunning(self):
        if not self._isSubProcessRunning():
            self._launch()
            self._outQueue = Queue.Queue()
            self._errQueue = Queue.Queue()
            self._startThreads()
            self._startExecutable()

    def _isSubProcessRunning(self):
        """
        Just checks if the thread is running.
        """
        # Check if child process has terminated. Set and return returncode attribute.
        if self._subproc.poll() is None:
            return True
        else:
            return False

    def __del__(self):
        if self._isSubProcessRunning() :
            self.exit()

def test():
    executable = '/net/serhom/home/cs/richard/Free_Lamp81/START_lamp -nws'
    prompt = "loaded ..."
    exitCommand = "exit"

    lamp = Communicate(executable, prompt, exitCommand)
    #time.sleep(0.2)
    output,errors = lamp.communicate('print, "Hello, Python"', waitTimeForTheCommandToGiveOutput=0.2)
    print "Output: ", output
    print "Errors: ", errors

    lamp.exit();

    output,errors = lamp.communicate('print, "Hello, Python"', waitTimeForTheCommandToGiveOutput=0.2)
    print "Output: ", output
    print "Errors: ", errors


if __name__ == '__main__':
    test()
```

AllToMantid: workspace

```
'''
Created on Oct 17, 2013

@author: leal
'''
import sys
# try :
#     sys.modules['mantid']
# except:
mantidBinDir    = '/opt/Mantid/bin'
sys.path.append(mantidBinDir)
from mantid.simpleapi import *

#from mantid.simpleapi import *
import numpy as np
import string
import random
import datetime

class Workspace(object):
    '''
    classdocs
    '''


    thisWs = None

    def __init__(self, wsName=None):
        '''
        Constructor
        '''
        if wsName is None:
            self.name = self._generateStringFromTime()
        else:
            self.name = wsName

    def _generateRandomString(self, prefix='ws_', length=4):
        return prefix+"".join(random.sample(string.letters*5,length))

    def _generateStringFromTime(self, prefix='ws_'):
        now = datetime.datetime.now()
        t = now.time()
        return prefix+"%02d%02d%02d"%(t.hour,t.minute,t.second)

    def _calculateEnergy(self,wavelength):
        h = 6.62606896e-34
        neutronMass = 1.674927211e-27
        meV = 1.602176487e-22
        e = (h * h) / (2 * neutronMass * wavelength * wavelength * 1e-20) / meV
        return e;

    def createFromData(self,data,xAxis,unitX="Wavelength",yUnitLabel='Counts'):
        """

        @param data: Assuming for now the data is a numpy 2D array
        @param outWorkspaceName:
        """

        # http://www.mantidproject.org/Extracting_And_Manipulating_Data

        # Add last bin to X
        lastBin = xAxis[-1] + (xAxis[-1]-xAxis[-2])
        xAxis = np.append(xAxis,lastBin)

        (nRows, nCollumns) = data.shape
        dataFlat = data.flatten() # convert to 1D

        xAxisClonedNRowsTimes = np.tile(xAxis,nRows)

        thisWs = CreateWorkspace(DataX=xAxisClonedNRowsTimes, DataY=dataFlat, DataE=np.sqrt(dataFlat),
                                         NSpec=nRows,UnitX=unitX,YUnitLabel=yUnitLabel,OutputWorkspace=self.name)



    def _valid(self):
        '''
        Make sure the ws exists.
        Don't know why mantid looses it...
        '''
        if self.thisWs is None:
            print "Warning: self.thisWs is None"
            self.thisWs = mtd[self.name]
            if self.thisWs is None:
                print "ERROR: self.thisWs is still None"
                return False
        return True

    def setProperties(self,propertyDic):
        """
        @param propertyDic: must be pair of string:string

        """
        if self._valid() == False:
            sys.exit(-1)

        r = self.thisWs.run()

        for key, value in propertyDic.iteritems():
            r.addProperty(key,value,True)


    def setAndCorrectProperties(self,propertyDic):
        """

        """
        if self._valid() == False:
            sys.exit(-1)
        r = self.thisWs.run()
```

```
import re
wavelengthRE = re.compile(".+Wavelength.+angstroms.+")
for key, value in propertyDic.iteritems():
    print key, ":", value
    if key is not None and value is not None and len(key) > 0 and len(value) > 0 :
        if wavelengthRE.match(key) is not None:
            r.addProperty("wavelength",float(value),True)
            r.addProperty("Ei",self._calculateEnergy(float(value)),True)
        else :
            try:
                r.addProperty(key,float(value),True)
            except:
                r.addProperty(key,value,True)
```

AllToMantid: lamp

```
'''
Created on Oct 17, 2013

@author: leal
'''

import nxs

class Lamp(object):
    '''
    This class will keep data from a object imported from lamp.
    I.e. This class holds a Lamp python object
    To date only data import from a nexus file is supported. See below: importNexus
    '''

    # Parameters accessible
    parameters = {}
    data = None
    xAxis = None
    yAxis = None

    def __init__(self):
        '''
        Constructor
        '''
        pass

    def importNexus(self,filePath):
        '''
        Import LAMP generated Nexus file.
        Lamp generates nexus file with the command:
            write_lamp, '/tmp/ricardo', w=2, format='HDF'
        where w is the workspace number
        '''
        nexusFileHandler = self._openNexusFile(filePath)

        self.data = self._readData(nexusFileHandler)
        self.xAxis = self._readData(nexusFileHandler,dataFieldName = 'X')
        self.yAxis = self._readData(nexusFileHandler,dataFieldName = 'Y')

        params = self._readData(nexusFileHandler,dataFieldName = 'PARAMETERS')
        self.parameters = self._parametersToDict(params)

        self._closeNexusFile(nexusFileHandler)

    def _openNexusFile(self,filePath):
        nexusFileHandler = nxs.open(filePath)
        nexusFileHandler.opengroup('entry1')
        nexusFileHandler.opengroup('data1')
        return nexusFileHandler;

    def _closeNexusFile(self,nexusFileHandler):
        nexusFileHandler.closegroup()
        nexusFileHandler.closegroup()
        nexusFileHandler.close()

    def _readData(self,nexusFileHandler, dataFieldName = 'DATA'):
        nexusFileHandler.opendata(dataFieldName)
        a = nexusFileHandler.getdata()
        nexusFileHandler.closedata()
        return a


    def _parametersToDict(self,params):
        """
        @param params: string
        """
        paramsAsListOfLists = [ [j.strip() for j in i.split('=')] for i in params.split('\n')]

        return dict(paramsAsListOfLists)

    def showSnapShot(self,filename):
        """
        It prints the snapshot in the Lamp nexus file
        There's no really use for these...

        @param filename:
        """
        f = nxs.open(filename)
        f.opengroup('entry1')
        f.opengroup('data1')
        f.opendata('SNAPSHOT')
        a = f.getdata()
        f.closedata()
        f.closegroup()
        f.closegroup()
        f.close()
        import matplotlib.pyplot as plt
        plt.imshow(a)
        plt.show()

    def _getParameter(self, desc):
        for p in self.parameters.keys():
            if desc.lower() in p.lower():
                return self.parameters[p]
    def wavelength(self):
        w = self._getParameter('Wavelength (angstroms)')
        if w is not None:
            return float(w)

if __name__ == '__main__':
    l = Lamp()
    l.importNexus('/tmp/ricardo_LAMP.hdf')
    import pprint
    pprint.pprint(l.parameters)
    pprint.pprint(l.parameters.keys())
    pprint.pprint(l.xAxis)
    pprint.pprint(l.yAxis)
    print l.data.shape
    #l.showSnapShot('/tmp/ricardo_LAMP.hdf')
```

```
print l.wavelength()
```

ReductionServer: main

```
#!/usr/bin/python

import bottle
from bottle import route
from bottle import response

import sys
import logging
import time
import signal


import config.config

import data.messages
from content.validator.filename import FileValidator
from query.handler import QueryHandler
from result.handler import ResultHandler
from status.handler import StatusHandler
from methods.handler import MethodsHandler

'''

Bottle reduction server

To old_test use curl:
-X GET | HEAD | POST | PUT | DELETE
Use curl -v for verbose

It assumes:
    - One server will run for a single instrument
    - Just a single file is handled by the server at the same time.
        - The submission of new file will invalidates the stored data of the previous one

'''



logger = logging.getLogger("server")


# Handle signals
def signal_handler(signal_, frame):
    logger.info("Server caught a signal! Server is shutting down...")
    logger.info("Killing running processes...")

    # TODO
    # Any cleanups needed

    time.sleep(0.1)

    logger.info("Server shut down!")
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
signal.signal(signal.SIGTERM, signal_handler)

@route('/', method=['GET','POST'])
def homepage_get():
    '''
    Home page:

    Open with a browser or:
    curl http://localhost:8080/
    '''
    logger.debug('Home page was requested.')
    return data.messages.Messages.success("Server is up and running.")


@route('/file/<numor:re:[0-9]+>', method='POST')
def fileHandler(numor):
    '''

    User can send a binary / ascii file or an url for a file location.

    To test:

    curl -v --noproxy '*' -X POST --data-binary @094460.nxs http://localhost:8080/file/094460

    curl -v --noproxy '*' -X POST --data "`pwd`" http://localhost:8080/file/094460

    '''

    logger.debug("Receiving file by HTTP POST with numor = %s" % numor)

    content = bottle.request.body.read()

    v = FileValidator(content)
    message = v.validateFile(numor)

    logger.debug(message)
    return message


#@route('/query/<numors:re:[0-9,]+>', method='POST')
@route('/query', method='POST')
def query():
    '''
    Get the query results. Sent json by the client should be of the form:
    { "method" : "theta_vs_counts", "params" : { "numors":[94460]} }
    '''

    content = bottle.request.body.read()
    logger.debug("RAW Query received: " + str(content))

    qh = QueryHandler(content)
    message = qh.process()
    logger.debug(message)
    return message
```

```
@route('/results/<queryId>', method=['POST','GET'])
def results(queryId):
    """
    Return the contents of localDataStorage has json

    Test:
    curl -X POST  http://localhost:8080/results/<queryId>
    """

    r = ResultHandler(queryId)
    message = r.getQuery()
    logger.debug(message)
    return message

@route('/resultszipped/<queryId>', method=['POST','GET'])
def resultszipped(queryId):
    """
    Return the contents of localDataStorage has json

    Test:
    curl -X POST  http://localhost:8080/resultszipped/<queryId>
    """

    r = ResultHandler(queryId)
    message = r.getQueryZipped()
    logger.debug("Zipped content! size = %d"%len(message))
    bottle.response.set_header('Content-Encoding', 'gzip')
    return message


@route('/status', method=['POST','GET'])
def status():
    """
    Returns data of queries
    """

    r = StatusHandler()
    message = r.getQueries()
    logger.debug(message)
    # Because the response is [...] it's not considered json
    response.content_type = 'application/json'
    return message

@route('/methods', method=['POST','GET'])
def methods():
    h = MethodsHandler()
    message = h.getAllMethods()
    logger.debug(message)
    return message

@route('/methodsavailable', method=['POST','GET'])
def methodsAvailable():
    h = MethodsHandler()
    message = h.getMethodsForThisInstrument()
    logger.debug(message)
    return message



def main(argv):
    # command line options
    from config.config import options

    # Launch http server
    bottle.debug(True)

    try :
        bottle.run(host=options.server, port=options.port)
    except Exception as e:
        #logger.exception("Web server cannot run: " + str(e))
        logger.error("Web server cannot run: " + str(e))

    logger.info("Server shutdown...")

if __name__ == '__main__':
    main(sys.argv)
```
