![A description...](data:None;base64,)

***Report on the d***evelopment of prototype software **(Task 3 D6.3)**

NMI-3 Workpackage 6 FP7/NMI3-II project number 283883

*Aug 29th, 2014 - R. Leal and E. Farhi (with input from members of the workpackage)*

*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](#__RefHeading__1154_21335345)

[Contributed Mantid loaders 2](#__RefHeading__1156_21335345)

[Contributed Mantid algorithms 3](#__RefHeading__1158_21335345)

[Contributed Mantid instrument definitions 4](#__RefHeading__1160_21335345)

[Contributed Mantid framework changes 4](#__RefHeading__1162_21335345)

[Other contributions: AllToMantid and reductionSever 5](#__RefHeading__1164_21335345)

[AllToMantid 5](#__RefHeading__2318_21335345)

[reductionServer 5](#__RefHeading__2320_21335345)

[Appendix: code produced 5](#__RefHeading__1166_21335345)

[LoadILL 6](#__RefHeading__1820_21335345)

[LoadILLAscii 12](#__RefHeading__1822_21335345)

[LoadILLIndirect 16](#__RefHeading__1824_21335345)

[LoadILLReflectometry 20](#__RefHeading__1826_21335345)

[LoadILLSANS 25](#__RefHeading__1828_21335345)

[LoadLLB 31](#__RefHeading__1830_21335345)

[LoadSINQFocus → LoadSINQ 35](#__RefHeading__1832_21335345)

[ConvertEmptyToTof 38](#__RefHeading__2043_21335345)

[CorrectFlightPaths 43](#__RefHeading__2045_21335345)

[DetectorEfficiencyCorUser 45](#__RefHeading__2047_21335345)

[SetupILLD33Reduction 48](#__RefHeading__2049_21335345)

[AllToMantid: communicator 56](#__RefHeading__2322_21335345)

[AllToMantid: workspace 58](#__RefHeading__2324_21335345)

[AllToMantid: lamp 60](#__RefHeading__2326_21335345)

[ReductionServer: main 62](#__RefHeading__2328_21335345)

## 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 reoported in our Task 1 and Task 2 documents, the Mantid project fulfills 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,w)* 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. The Mantid contributions have been included in the project, and should be maintained by the Mantid develeoppers in the future.

## 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 developped in parallel with this workpackage, 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 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) |
| SANSSensitivityCorrection | Perform SANS sensitivity correction. | SANS (minor modification) |
| 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 attaxched 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 catenated 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 acqiosition 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 Agorithm takes as imput 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 and iFit.

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](http://www.json.org/) 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

| 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

| 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)