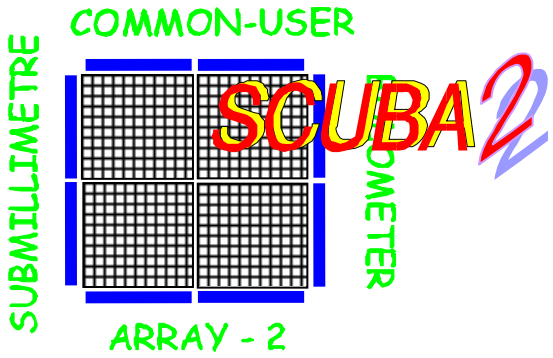


SCUBA-2 FTS Project Office
University of Lethbridge
Physics Department
4401 University Drive
Lethbridge, Alberta
CANADA
T1K 3M4

Tel: 1-403-329-2771
Fax: 1-403-329-2057
Email: brad.gom@uleth.ca
WWW: http://research.uleth.ca/scuba2/

Document Title: **FTS-2 DR Java Package**

Document Number: **SC2/FTS/SOF/004**

Issue: **Version 1.2**

Date: **2 November 2006**

Document Prepared By:	B. Zhang FTS Software Engineer	Signature and Date:	02/11/06
Document Approved By:	D. A. Naylor FTS Project Lead	Signature and Date:	02/11/06
Document Released By:	J. Molnar Canadian Project Manager	Signature and Date:	02/11/06

Change Record

Issue	Date	Section(s) Affected	Description of Change / Change Request Reference / Remarks
0.1	04/04/06	All	First draft
1.0	30/10/06	All	CDR version
1.1	31/10/06	2.2	Added deglitching section
1.2	2/11/06	2	Minor formatting fixes

Contents

1. Introduction

This document describes the Java package created for the FTS-2 DR engine processing code. Java was chosen as the engine language due to the heritage of the Herschel SPIRE project, from which many of the FTS-2 algorithms are derived. The Java code processes interferogram data into spectral data, and consists of algorithm classes for interpolating the interferogram data onto a regular grid, phase correction, and Fourier transformation (FFT).

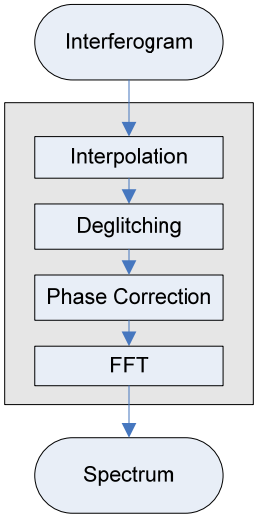
In the case of an ideal interferometer, an interferogram is a real and even function of the optical path difference (OPD) between the interfering radiation beams. In the ideal case, only one side of the interferogram must be recorded and the spectrum can be calculated using a single-sided Fourier cosine transform.

In practice, real interferometers produce interferograms with a path difference that varies with the frequency of the radiation, due to optical, electronic or sampling effects. Further, the discrete sampling might not include the zero path difference (ZPD) position. In this case, the interferograms are no longer symmetric and the spectra cannot be retrieved using a simple single-sided transform.

The Java code described in this document corrects the asymmetry of an interferogram (so that a single-sided transform may be used), resamples the interferograms onto an even OPD grid (so that a FFT can be used), applies a phase correction function, and then calculates the final spectra.

2. The FTS Java Package

A non-ideal (phase-distorted and irregularly-sampled) interferogram must be re-sampled evenly (or interpolated) and phase-corrected before one-side cosine FFT can be applied. The following flowchart shows how to compute a spectrum from a non-ideal interferogram.



2.1. Interpolation

In the FTS-2 Java package, a natural cubic spline algorithm (i.e., the second derivatives of each polynomial is set to zero at the endpoints) is applied for interpolation. This algorithm is implemented by **ca.uol.aig.fts.fitting.CubicSplineInterpolation**. The major constructor of this Java class is:

```
CubicSplineInterpolation(double[] x_orig)
```

where `x_orig` is an array representing the OPD of an irregularly-sampled interferogram.

This Java class has only one method:

```
public double[] interpolate(double[] y_orig)
```

where `y_orig` is an array representing the intensity values of an irregularly-sampled interferogram. The return value is an array representing the intensity values of the new evenly-sampled interferogram.

This algorithm is optimized for the case where multiple interferograms share the same `x_orig`.

2.2. Deglitching

The deglitching algorithm is implemented by the Java class:

ca.uol.aig.fts.deglitch.Deglitching

This Java class has only one constructor:

```
Deglitching(int dsSize, int index_ZPD)
```

where, `dsSize` is the half size of the double-sided interferogram and `index_ZPD` is the index number of the ZPD.

It also has only one public method:

```
public void deglitch(double[] ifgm, int deglitching_flag)
```

where `ifgm` is an input interferogram to be deglitched, `deglitching_flag` is the deglitching flag. Possible values for `deglitching_flag` are:

- 1 - only deglitch the core part of the interferogram
- 2 - only deglitch the tail part of the interferogram
- 3 - deglitch both of the core part and the tail part of the interferogram
- other values - no deglitching

2.3. Phase Correction

The phase correction algorithm is implemented by the Java class:

ca.uol.aig.fts.phasecorrection.PhaseCorrection

The major constructor of this Java class is:

```
PhaseCorrection(int dsLength, int ssLength, int phaseFittingdegree, int
pcfSize_h, double weight_limit, int ZPD_index, int interferogram_len,
                double wn_lBound, double wn_uBound)
```

Parameters	Comments
`dsLength`	Half of the length of the double-sided interferogram (points).
`ssLength`	Length of the single-sided interferogram (points).
`phaseFittingdegree`	The degree of the polynomial used to fit the phase.
`pcfSize_h`	Half of length of the phase correction function (points).
`weight_limit`	Amplitude threshold below which phase points will be neglected in the phase correction.
`ZPD_index`	The index number of the ZPD in the interferogram.
`interferogram_len`	The total length of the original interferogram (points).

`wn_lBound`	The fractional position (0 – 1) in the phase array below which phase data will not be included in the phase fitting. The lowest wavenumber used in phase-fitting is wn_lBound x dsSize x Nyquist.
`wn_uBound`	The fractional position (0 – 1) in the phase array above which phase data will not be included in the phase fitting. The highest wavenumber used in phase-fitting is wn_uBound x dsSize x Nyquist.

The major method of this Java class is:

```
public double[] getInterferogram(double[] compositeInterferogram,
                                 double[] phaseFitting_stderr)
```

where `compositeInterferogram` is the original interferogram (with a long single-sided part of length `ssLength` and a short double-sided part of length is 2 x `dsLength`) and `phaseFitting_stdderr` is the standard deviation of the phase fitting. The return value is the phase-corrected single-sided interferogram with length (`ssLength`+1).

2.4. FFT

A Cosine FFT is used to obtain the spectrum from a phase-corrected single-sided interferogram. The Cosine FFT is implemented in the Java version of the **fftpack** package as: **ca.uol.aig.fftpack.RealDoubleFFT_Even**.

The constructor of this Java class is:

```
RealDoubleFFT_Even(int n)
```

where `n` is the size of the wavenumber table to be constructed, and can be any positive integer. When (n-1) can be factored by small numbers (4, 2, 3, 5), this FFT transform is very efficient. The size of the real periodic sequence corresponding to this real even sequence is 2(n-1).

This Java class has two methods: **ft** and **bt**. The forward cosine FFT transform, **ft**, has the following form:

```
public void ft(double[] x)
```

where `x` is the array to be transformed. After the FFT, `x` contains the transform coeffients. The backward cosine FFT transform, **bt**, has the following form:

```
public void bt(double[] x)
```

where `x` is the array to be transformed. After the FFT, `x` contains the FFT transform coeffients, i.e., the spectrum.

RealDoubleFFT_Even has a constant, *norm_factor*, used to normalize the FFT. This normalization is required since a call of the forward transform (**ft**) following by a call of backward transform (**bt**) will multiply the original input sequence by *norm_factor*:

```
x = bt(ft(x))/norm_factor
```

The forward transform of **RealDoubleFFT_Even** is identical to its backward transform.

2.5. Example

In this section, a simple example, **TestDRPipeline.java**, is given to illustrate how to call three Java classes mentioned above efficiently. In the following example, an interferogram cube, `ifgm_cube(x, y, z)` (`x, y` are the indices of pixel positions, `z` is the index of points in an interferogram (x, y)) is used as input and `spectrum_cube(x, y, z)` as output. Meanwhile, all interferograms in `ifgm_cube` share the same OPD grid. (Note that the obliquity effect will require spectral calibration for off-axis pixels.)

```
import ca.uol.aig.fts.fitting.CubicSplineInterpolation;
import ca.uol.aig.fts.phasecorrection.PhaseCorrection;
import ca.uol.aig.fftpack.RealDoubleFFT_Even;
import ca.uol.aig.fts.deglitch.Deglitching;


public class TestDRPipeline
{
      public static void main(String[] args)
      {
           int arrayLength=40, arrayWidth=32;
           int dsSize=300, ssSize=5001;
           int pcfSize_h=80, fittingDegree=2;
           double weight_limit=0.05;
           double zpd_value = 400.0;
           double wn_lBound = 0.05;
           double wn_uBound = 0.95;
           int deglitch_flag = 3;

           double[] opd = new double[dsSize+ssSize-1];
           double[][][] ifgm_cube =
                 new double[arrayWidth][arrayLength][dsSize+ssSize-1];

           /* make data for ifgm_cube */
           for(int i=0; i<dsSize+ssSize-1; i++)
               opd[i] = i;

           for(int i=0; i<arrayWidth; i++)
              for(int j=0; j<arrayLength; j++)
                 for(int k=0; k<dsSize+ssSize-1; k++)
                    ifgm_cube[i][j][k] = i+j+k;

           TestDRPipeline testDR = new TestDRPipeline();
           double[][][] spectrum_cube =
                 testDR.DRPipeline(ifgm_cube, opd, pcfSize_h,
                            dsSize, ssSize, fittingDegree,
                            weight_limit, zpd_value,
```

```
                        wn_lBound, wn_uBound, deglitch_flag);
    }

    double[][][] DRPipeline(double[][][] ifgm_cube, double[] opd,
                   int pcfSize_h, int dsSize, int ssSize,
                   int fittingDegree, double weight_limit,
                   double zpd_value, double wn_lBound,
                   double wn_uBound, int deglitch_flag)
    {

        /* instantiate Interpolation */
        CubicSplineInterpolation csi2fts = new
                      CubicSplineInterpolation(opd);

        Int index_ZPD = csi2fts.getIndex_ZPD(zpd_value);
        int interferogram_len = csi2fts.getInterferogramLength();

        /* instantiate Phase-Correction */
        PhaseCorrection pc2fts =
               new PhaseCorrection(dsSize, ssSize, fittingDegree,
                               pcfSize_h, weight_limit, index_ZPD,
                               interferogram_len,
                               wn_lBound, wn_uBound);

        int pc_dsSize = pc2fts.get_dsLength();
        /* instantiate Deglitching */
        Deglitching deglitch2fts =
                 new Deglitching(pc_dsSize, index_ZPD);

        /* instantiate RealDoubleFFT_Even */
        int new_ssSize = pc2fts.get_ssLength();
        RealDoubleFFT_Even cosfft = new
                     RealDoubleFFT_Even(new_ssSize+1);

        int arrayWidth = ifgm_cube.length;
        int arrayLength = ifgm_cube[0].length;

        /* interpolate the interferograms */
        double[] single_ifgm;
        double[][][] ifgm_interp =
                 new double[arrayWidth][arrayLength][];

        for(int i=0; i<arrayWidth; i++)
           for(int j=0; j<arrayLength; j++)
           {
               single_ifgm = ifgm_cube[i][j];
               ifgm_interp[i][j] = csi2fts.interpolate(single_ifgm);
           }

        /* deglitch the interferograms */
        for(int i=0; i<arrayWidth; i++)
           for(int j=0; j<arrayLength; j++)
           {
                deglitch2fts.deglitch(ifgm_interp[i][j],
                                deglitch_flag);
           }
        /* phase-correct interferograms */
```

```
            double[] pc2fts_stderr = new double[1];
            double[][][] ifgm_pc = new
                      double[arrayWidth][arrayLength][];
            for(int i=0; i<arrayWidth; i++)
               for(int j=0; j<arrayLength; j++)
               {
                   ifgm_pc[i][j] =
                          pc2fts.getInterferogram(ifgm_interp[i][j],
                                                  pc2fts_stderr);
               }

            /* FFT of interferograms */
            for(int i=0; i<arrayWidth; i++)
               for(int j=0; j<arrayLength; j++)
                    cosfft.ft(ifgm_pc[i][j]);

            return ifgm_pc;
         }
}
```

3. FTS-2 Java Classes

Besides the general Java classes for FTS processing given above, there are four Java classes specific to the FTS-2 system:

- **ca.uol.aig.fts.io.NDFIO**
- **ca.uol.aig.fts.display.QuickLookXY**
- **ca.uol.aig.fts.message.Drama2FTS**
- **ca.uol.aig.fts.message.SOAP2FTS**

3.1. NDF I/O

The **NDFIO** class is used to read a FTS-2 specific NDF file which contains an interferogram cube and create a corresponding spectrum file. The constructor is:

```
public NDFIO(String interferogramFile, String spectrumFile)
```

where `interferogramFile` is the absolute path of an input interferogram NDF file, and `spectrumFile` is the absolute path of the spectrum file corresponding to `interferogramFile`.

This class has the following methods:

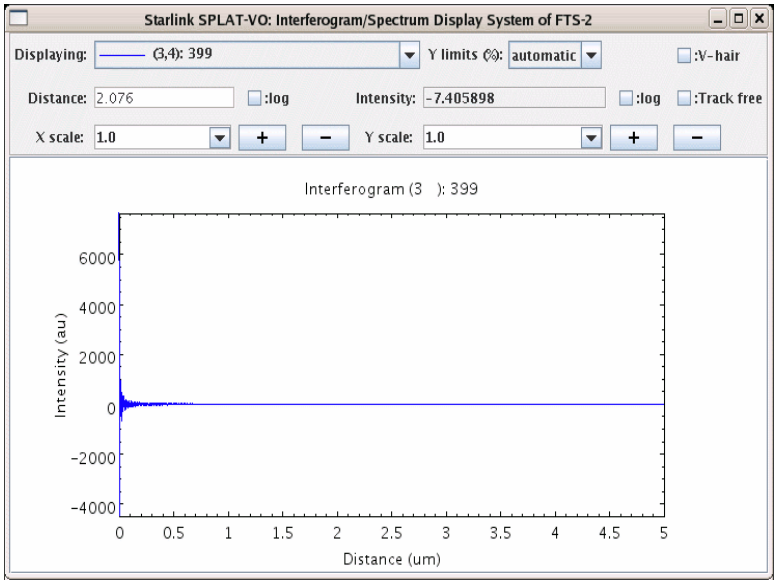
Method	Description
`void` **`closeSpectrum`**`()`	Close the handle of the spectrum file
`int` **`get_arrayLength`**`()`	Get the size of the array in y-axis
`int` **`get_arrayWidth`**`()`	Get the size of the array in x-axis
`long[]` **`get_ifgmCubeShape`**`()`	Get the dimension of the interferogram file
`long` **`get_ifgmCubeSize`**`()`	Get the total number of the data in the interferogram cube.
`String` **`get_ifgmType`**`()`	Get the data type of the interferogram cube. Possible values: _UWORD, _WORD, _INTEGER, _REAL, _DOUBLE.
`int` **`get_npoints_ifgm`**`()`	Get the number of the data of one interferogram
`java.lang.Object` **`getInterferogram`**`()`	Get the whole interferogram cube as 1-D data array stored in Fortran format
`double[]` **`getInterferogram`**`(int indexOfWidth, int indexOfLength)`	Get a specified interferogram. indexOfWidth is the index of the array pixel in x-axis starting from 0; indexOfLength is the index of the array pixel starting from 0.
`double[]` **`getMirrorPos`**`()`	Get the mirror position (or OPD) from the interferogram file

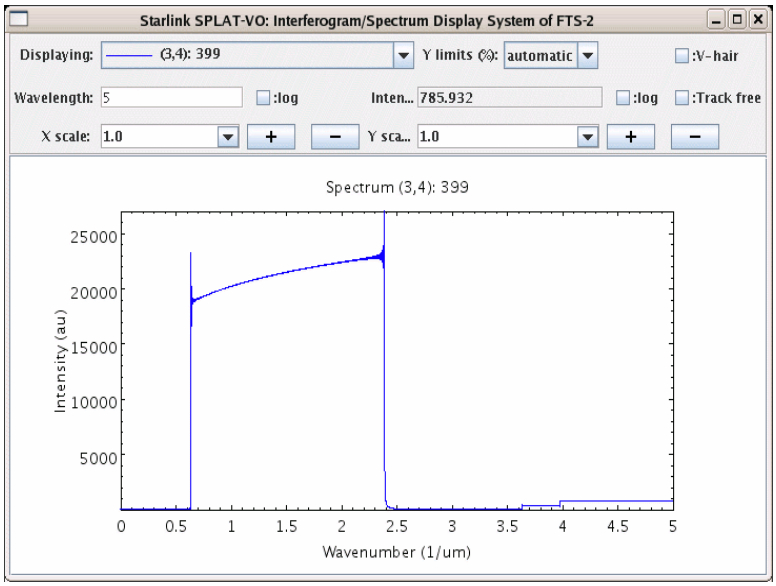
`void saveSpectrum(double[] spectrum, long[] dims)`	Save the spectrum cube to the spectrum file. The spectrum cube is stored in 1-D Fortran data order. dims is the dimension information of the spectrum. dims[0] is the size of the first dimension of the interferogram, dims[1] is that of the second dimension, and dims[2] is that of the third dimension.
`void saveSpectrum(float[] spectrum, long[] dims)`	Save the spectrum cube to the spectrum file. The spectrum cube is stored in 1-D Fortran data order. dims is the dimension information of the spectrum. dims[0] is the size of the first dimension of the interferogram, dims[1] is that of the second dimension, and dims[2] is that of the third dimension.
`void saveSpectrum(int[] spectrum, long[] dims)`	Save the spectrum cube to the spectrum file. The spectrum cube is stored in 1-D Fortran data order. dims is the dimension information of the spectrum. dims[0] is the size of the first dimension of the interferogram, dims[1] is that of the second dimension, and dims[2] is that of the third dimension.

Due to the symmetry of an even real interferogram, the corresponding spectrum is also even and real. Therefore, only the first half of its spectrum is stored in `spectrumFile`. For those who like to think in terms of positive and negative frequencies, this means that only the zero frequency and positive frequencies are stored in `spectrumFile`.

3.2. Quick Look (optional)

The Java class **ca.uol.aig.fts.display.QuickLookXY** can be used to display 1-D x-y curves (e.g., an interferogram or spectrum curve). QuickLookXy is based on the SPLAT (Starlink Spectral Analysis Tool) display system. The following two pictures show an interferogram and a spectrum displayed with this tool.





The constructor of **QuickLookXY** is:

```
QuickLookXY(String x_Symbol)
```

where `x_Symbol` is the symbol of x-axis.

QuickLookXY has two methods:

```
showInterferogram(double[] x, double[] y, String xUnit, yUnit, String
                  identifier)
```

and

```
showSpectrum(double[] x, double[] y, String xUnit, yUnit, String
                                identifier)
```

where `x` contains the x-axis values, `y` contains the y-axis values, `xUnit` is a string representing the data units for the x-axis, `yUnit` is a string representing the data units for the y-axis, and identifier is a string used to identify different curves.

3.3. FTS-2 Messaging System

The FTS-2 DR engine supports two different communication mechanisms within the ORACDR pipeline. The 'FTS-2 Data Reduction Engine' document, SC2/FTS/SOF/001, details their usage.

References

[1] Numerical Recipes, http://www.numerical-recipes.com/.
[2] JFFTPack and relevant documents, http://cm.bell-labs.com/netlib/fftpack/.
[3] StarJava, http://ww.starlink.rl.ac.uk/java/java.htm.
[4] SPLAT (Spectral Analysis Tool), http://www.starlink.ac.uk/splat.
[5] FTS-2 Data Reduction Engine, version 2.0.