1 Introduction

This describes a Python program wrapper for the CCP4 Program Scala (Evans, P "Data Reduction" in Proceedings of 1993 CCP4 Study Weekend.) This wrapper provides most of the commonly used functionality implemented in Scala.

2 Use Cases

2.1 Simple 1: Scale and Merge Data

The simplest implementation of the Scala wrapper is to simply take data from Mosflm and merge to a "standard" mtz file. This will need to perform the following operations:

- Apply a resolution limit.
- Apply a scaling model.
- Apply an error model.
- Add project, crystal, dataset information.
- Raise an error sensibly if the input data are not sorted; are not MTZ format; do not exist...

Use case test data exists for this in the XIA core test data repository. This simple version will do something like:

```
#!/bin/bash
scala DNA scala.stf \
HKLIN 12287_1_E1_scaled_sorted.mtz \
HKLOUT 12287_1_E1_scaled.mtz << eof
USECWD
NAME PROJECT default CRYSTAL default DATASET default
run 1 all
sdcorrection full 2.000000 0.095000 partial 2.000000 0.085000
resolution high 1.53937314091
anomalous on
cycles 20
scales rotation spacing 5.0 secondary 6.0 bfactor on tails
eof
```

2.2 Simple 2: Just Merge

This implementation would take data previously scaled by an outside program (for instance XSCALE; SCALEPACK) and merge the reflections to provide scaling statistics and a merged reflection file. This will need to provide the following operations:

- Apply a resolution limit.
- Add project, crystal, dataset information.

Tools to transform the input data from whatever the originating format is would be provided externally, and must be used.

FIXME/to-do: Test data for this will need to be added. Make it so or generate the test data during the implementation of the XDS/XSCALE wrappers.

This will do something like:

```
#!/bin/bash
combat hklin SCALED.HKL hklout SCALED.HKL.tmp1 << eof
scale 0.02
input XDSASCII
pname default
dname xds
eof

#!/bin/bash
sortmtz hklin SCALED.HKL.tmp1 \
hklout 12287_1_E1_scaled_scala.mtz.tmp2 << eof
H K L M/ISYM BATCH
eof

#!/bin/bash
scala hklin 12287_1_E1_scaled_scala.mtz.tmp2 \
hklout 12287_1_E1_scaled_scala.mtz \
dnaout xds_scala.stf << eof
run 1 all
scales constant
initial unity
anomalous on
eof
```

This is probably best implemented as a separate method - e.g. s.merge() in place of s.scale().

2.3 More Complex 1: Merging Data, 2 Passes

Two datasets processed with Mosflm, appropriately REBATCHED and SORTED. The resulting MTZ file will need to be scaled and merged appropriately to give reasonable merging statistics from the two or more runs. This will require:

- Apply resolution limits.
- Apply a scaling model.
- Apply error models per run.
- Add project, crystal, dataset information.

The error models are best decided by scaling the data separately then recycling the parameters.

2.4 More Complex 2: Scaling MAD Data

More than one data sets collected from the same crystal but at different wavelengths. The data will need to be scaled separately but used together for building the absorption model etc. This is much more complex.

This will produce a number of output reflection files.

This will further need the reflections to be appropriately sorted and merged together in the file, with sensible project/crystal/dataset information in advance of the processing.

FIXME this needs to be specified.

Test data for this exist within the XIA core source tree.

3 Implementation

3.1 Notes

To achieve a decent structure to this I may be better off implementing a system based around data sets rather than files. This will be OK because any data coming from outside (XDS, SCALEPACK) will have to have been passed through COMBAT first and will hence have the appropriate information in.

So, for instance

```
addDataset(reflection_file,           # perhaps this should be part of
                                      # the setHklin() method?
           batches = (start, end),    # inclusive
           project = x,               # these should be set to
           crystal = y,               # sensible values to help
           dataset = z,               # the wrapper decide what
           id = 'low_res_pass')       # is what

=> run 1 (say)

addError_model(id = 'low_res_pass',
               sd_partial = {'Fac':1.0,
                             'Add':0.02,
                             'B':15},
               sd_full = {'Fac':1.0,
                          'Add':0.02,
                          'B':15})
```

may be a sensible way to make this work. The calling program can then assign id's appropriately as well as the standard error correction parameters.

In MAD scaling it's probably the case that all wavelengths should have the same standard error parameters, so this is a fairly efficient way of recycling this information.

I'm not sure how likely it is that someone will want to scale multiple wavelengths and multiple sweeps per wavelength, but in this case I guess that anything is possible (there are JCSG examples like this - identify them.)

To-do: ID some interesting JCSG examples for this.

From the project/crystal/dataset information assumptions WILL be made about what should be done. If two datasets have the same project/crystal/dataset hierarchy then they should be merged together; if they have different wavelengths recorded in the reflection files then an exception will need to be raised.

Note further that it is not possible to input more than one reflection file into scala - in all cases above the data set reflection file should be the same

- anything else will raise an exception. Perhaps this aspect of the input should be relegated to the default setHklin() method.

However this kind of input would be very helpful for CAD, where this kind of thing wants to happen (though this could be at a higher level.)