# 1 Introduction

This document describes the principles and structure behind the xia2dpa data model. This is necessary because the problem is a complicated one. The basic principle here is to have a global "data repository" which is structured enough to have a hierarchy of data but simple enough that a value of something can be found easily.

This structure will need to have a few basic properties:

- Objects have two "parts" - an immutable identity and a set of defined properties which may be allowed to vary with time.
- Changes to objects should be recorded as updates, with earlier instances being kept. An example follows.
- Getting a mutable property for an immutable object will delegate the getting to an immutable child.

The example which follows is:

```
d = Dataset('12287_1_E1_001.img') -> Populate identity from the image
                                     headers for this data set. Initialise
                                     metadata about this dataset, and
                                     record the creation epoch.

d.index() -> Autoindex the data set. This task will be delegated to an
             autoindexer, which will take as an argument the dataset object.
             The results will be recorded in a list of autoindexing
             solutions and the latest instance in this list returned.
             This method will also check for assertions which may be
             relevant, for instance that the lattice is monoclinic.
             This assertion will also have an associated epoch. If ...

d.getCell() -> Is called and the assertion for the lattice is more recent
               than the source object that getCell gets the information from,
               the source object (in this case the indexing solution) needs
               to be reevaluated. This will mean that any further processing
               based on these results may need to be repeated.
```

The upshot of this is that if I autoindex the data set, process and find in cell refinement that the refinement breaks (or process in triclinic, and check the point group) I can assert that the lattice is something different. The next "get" method will then verify that it's information is up-to-date and if not will KNOW how to make it so.

This is going to get complicated, but is a fascinating way of working. It will mean that the knowledge on how to update objects will have to be delegated to the objects.

This comes back to the overarching idea that the main() routine in this could almost look like:

```
processing_results = Dataset('12287_1_E1_001.img').getProcessing_results()

... or almost ...

structure = Model(sequence = 1vpj.pir,
                  phases = Phase(Dataset({frames:['infl_001.img',
                                                  'lrem_lr_001.img',
                                                  'lrem_001.img'],
                                          id:[(0.9790, fp, fpp),
                                              (1.0002, fp, fpp)]}))
                  ).getStructure()
```

... taking this to the obvious conclusion, the objects would have ONLY get methods - everything else would be passed in through the constructor, and all actions would be implied by the get methods. Note that "private" methods would be needed in order to implement the result discovery delegation but this would be relatively doable.

Time to get a second brain fitted then. This is beginning to look a little like hard-core C++ programming.

This then means that the whole architecture is almost programmed in a functional manner[1]. Cool. It also means that the schema for the objects is of relatively little interest, though it would be handy to have some lightweight objects for handling this kind of information.

Basic idea here is everything works by lazy evaluation - only compute the result when asked for it, not when you're asked to compute it.

### 1.1 Hierarchy

This principle really works when ideas of hierarchy are included. For instance, the first pass at processing the first data set (by definition the reference) will result in one unit cell. This will then be set as the "globally correct" one until more information is available. When another set at this wavelength is processed, a weighted value for the unit cell may be derived from both sets. When combining all of the data, an overall unit cell may be computed from all of the available data sets.

This means, by implication, that each data processing "run" will have an associated local instance of a global data store. When it comes to merging or combining data sets, a further data store will be necessary. Does this mean that each stage is considered as it's own project, resulting in an idea that the global project data repository, referred to above, is some kind of weighted average of all of the local data repositories. Think of this like a tree-code.

## 2 Items of Interest

### 2.1 User Input

At the beginning, all that is known is what the user has passed in on the command line. From this a small number of simple things can be derived. In the example above, an example is the relationship between datasets and f values - e.g. wavelength for an image is read from the header. This is then used to associate different sweeps collected at the same wavelength and so on...

So - an object is needed to hold the information passed in by the user. Frameworks are then needed to communicate this information to the con-

[1] http://en.wikipedia.org/wiki/Functional_programming

structors as described above, perhaps as part of a dictionary which the constructor can interrogate as necessary.

The following items will be minimally necessary to make this work:

- Frames to use, defined by a frame from the set.
- ?Correct beam centre.
- ?Wavelength, f', f" values if available.

### 2.2 Learning Things

From the basic information derived from the command line, some extra information can be learned. For example:

- Lattice, unit cell.
- Spacegroup?
- Resolution.

At any stage any derived information is a hypothesis. There will be degrees of reliability associated with each of these, and when asked for the most recent (by definition most reliable) instance should be returned.

## 3 Defined Objects

### 3.1 Object

The class Object defines some really basic stuff, and should be inherited from by all objects. In particular it defines an identity which will enable sorting by creation epoch. Since all data will be defined at construction time, and identities are immutable, this *should* be safe!

Also - added a mutex to this to allow bottom-up implementations of threading.

Also - add a list of output to each object, to allow recording what each object actually does. When overall output is wanted, each object could print it's stuff.

### 3.2 Sweep

A sweep object defines a set of continuous frames. This can therefore be characterized by phi start and end values, oscillation width, exposure time, distance, wavelength, template and exposure epoch range. The if the epoch is NULL then the template will be used for sorting, otherwise start epochs will be most important.

FIXME the frame identification etc. defined in Dataset should probably be delegated to Sweep, and then contained in Dataset. All of the "expertise" should then be obtained through this interface.

In fact the constructor for this should include the header reading functioality so that we can be sure that the contents of the object are correct - also simplifies the interface and ensures that delegation is correct.

Sweeps should really come from a sweep factory, which should be accessed from the Dataset object - the Dataset can then decide what is appropriate to do with the sweeps.

=¿ define a SweepFactory in the Sweep object which will return a list of sweeps. However, there is one niggle - I want sweeps to be able to update themselves just in case more frames have appeared since they were last used. This means that there needs to be a higher level of sweep owner to manage all of this - OR - a sweep has to contain subsweeps or something, with those being arranged by collection date. Could "identify" sweeps by the first image (since I will assume image numbers always increment during collection.) That way the identity will not change, and so they can be picked out.

And it also means that a SweepFactory is possible... Finally, assume that image headers do not change! They can therefore be cached in the Printheader class, which is useful because it will really speed things up when there are multiple reads

FIXME: Add a feature to identify the detector class and mode, e.g. "ADSC Quantum 315 2x2 binned" and also a short code like "q315-2x2".

### 3.3 Dataset

# 4 Delegation

Objects like dataset may have well defined methods for performing tasks. However, something which could be fun would be to delegate this via `__getattr__` and a module registry to allow any arbitrary object to have a punt at performing an operation. Would this be safe?

Or is it a better idea to just have:

```
d = Dataset(...)
d.getLatticeInfo() -> delegate via IndexerFactory to get an implementation,
                      then return the result...
```

# 5 Interfaces

## 5.1 Thoughts

Since some programs present more than one interface, is it appropriate to inherit from base classes which represent these interfaces as well as the "Driver"? That would ensure that when you say class X implements indexer you would be certain that it does, because the indexer interface would be the only way to get to the functionality.

Yes, this is probably a good idea. The only problem is then to ensure that there are no name clashes between interfaces which might me multiply represented, e.g. indexer and integrater for Mosflm & XDS.

For information, multiple inheritance does work in Python.

An interesting question is how to handle the directory, template information - since almost all (or all?) data processing interfaces will need this information is it also better to inherit from (or decorate?) a Driver to handle this information "invisibly"? Probably. Then all programs which require diffraction images should use this information as the sole way of getting to the images. Since this is hidden invisibly by the CommandLine singleton there shouldn't be any big problems.

These have just been moved from /Interfaces to /Schema/Interfaces. 10/JUL/06.

### 5.2 Frame Processor

This is an interface which includes all of the information which may be needed by something which handles diffraction images. This includes:

- Beam position.
- Wavelength.
- Distance.
- Template.
- Directory.
- Header information e.g. width, pixel size.

This should be defined as a basic decorator in the same way that the CCP4 decorator works. This will give a little extra work for a lot of extra benefit. This will further mean that this goes into the xia2core definition, which makes it more simply available to the DC module.

[FIXME this section is now to go into xia2core documentation]

These would well suit a decorator if it didn't mean that the other decorators wouldn't work - looks like we're better off simply working by multiple inheritance and ducking when things go strange.

[UNFIXME this now needs to stay here!]

Added option to initialize the information from a constructor which takes an image file - this seems to make sense to me. Haven't made that aspect of the interface public, don't know whether I should.

## 5.3 Indexer

An indexer should take images to index with (either as a list of a block) perform indexing and provide the results in a useful fashion. The form of the results should be an orientation matrix, unit cell, lattice and an estimate of the mosaic spread. The refined beam position should also be returned.

Inputs should be the lattice (optionally), unit cell (optionally).

Ok, more thoughts, based on XDS, Mosflm, Labelit & d*TREK. This is what we need to be able to take as input:

- Frame processor information above. n.b. that this includes the distance, wavelength &c.
- Lattice; unit cell
- Input images to use - as a list of wedges[2]

Now for the outputs. The output in all cases should be the lattice, cell, mosaic and so on. Need to implement some way to "hide" extra information, for example the mosflm orientation matrix. In particular it would be useful to be able to share this kind of information in pipelines where we want to use e.g. only XDS. Maybe pass an "indexing information" bucket, where the information may or may not be - if it's not there then the *next* application will need to know how to regenerate the missing from the available information e.g. the unit cell.

Thought/FIXME: Shouldn't it be down to the indexer implementation to decide what images it wants to use for indexing - for instance d*TREK can make use of a couple of small wedges of data... - this is also missing the point of delegation. However, all indexers will need to be able to select from a list of images, so a sweep definition will need to be included in the input.

If list == NULL then decide; else use user defined list. This is best because it allows delegation of the selection to someone who knows & cares. To enable this, I have added a list of images to the FrameProcessor interface. This helps, because it means that you can look at the header of the first image and the other images to make your selection.

Update 4/AUG/06: Just changed the interface to use get_this set_that rather than the camelCase version - this is more tidy, but it may get confusing because many of the other objects still use camel case. I think there is some logic here, but I can't put it into words.

## 5.4 Architecture

Thought: An integrater takes as input an indexer, or it makes it's own. Can that fly? E.g. passes in the indexer because that contains all of the stuff

[2]This allows for all cases - if the wedges are written as a single number, use that, else use min(list) to max(list).

that the integrater needs to work, which could in turn be passed to a scaler to do it's funky stuff.

Interesting idea, no idea if it'll fly! Esp. because this could be a little recursive with things like Mosflm which implements indexer and integrater. Hmm.... This should be doable, but may be more than a little mysterious!

## 5.5 Integrater

The integrater interface will:

- [optionally] reindex if indexing solution for this program not part of indexer payload.
- [optionally] perform local cell refinement if the program uses this, and if the user has not asked us to be *fast*.
- Actually perform the integration, to a defined resolution limit [optional] and over a defined range of frames [optional: default to all.]
- [optionally] refine the integration parameters c/f mosflm gain and repeat the integration. Note that this is appropriate for Mosflm, XDS & d*TREK in their own ways.

### 5.5.1 Thoughts on the Output

Now looking at implementing an Integrater based on XDS, and it is therefore time to think about

- What information we want.
- How we want it displayed (pixels, mm?)
- What the coordinate frame should be for printing this stuff.

These are three interesting problems. Clearly knowing what we want first is critical, though this should be available in some form from all of the integration packages which will be considered (Mosflm, XDS, d*TREK, denzo, XGEN perhaps?)

In terms of providing an abstract interface which has enough information, there is a problem in that some of the programs have very different output to others. However, from a user perspective I guess all that is important is the final refined unit cell and the knowledge that xia was hapy, or not, with the integration of each image. Perhaps simply printing up "." for images it is happy with and "*" for those that it is unhappy with would be enough, along with perhaps some diagnostic information. This could more easily be written for a variety of different data reduction programs, and gets a long way from the very picky log file approach. Though I need to be very careful in integration that the verbatim log files are available for those who want them.

This is an interesting way to go.

### 5.6 Scaler

This is where things start to get complicated... The scaling tends to have a lot of program specific options in it, which presents this situation:

- We do not have a "standard" scaler interface, - or -
- We do have a standard scaler interface, but *hide* all of the options and encode expertise therein to cope with this.

Clearly the latter suggestion is sound. However, is there a good reason why a scaling program wrapper could not do both? That is, present an interface as a scaler BUT ALSO the program specific interface options? An interesting idea...

What is needed for a general Scaler interface? Input, output files (which will map onto HKLIN, HKLOUT for Scala) [sorting should be implicit, if not already sorted]. Must be also able to cope with

- Multiple sweeps for one wavelength which need to be merged.
- Multiple wavelength data which need to be scaled together to get the best signal out.

Initially I want to be able to work this with XDS/XSCALE, Mosflm/Scala with (maybe) the d*TREK package... think about adding denzo/scalepack at some point in the future...

### 5.7 Substructure Finder

### 5.8 Phase Computer

## 6 Input & Output Description: The .xinfo Files

At the moment I can identify the following as needing to come in the inputs:

- A list of existing data sets, and which crystals they have come from, and if the header is not populated with the collection date, that information.
- The appropriate f', f" values for each crystal & wavelength.
- Corrections to header information, in particular the beam centre.

- The information about the sample, is it SeMet, what heavy atoms do we expect, what is the sequence?

The first three of these can be collected during data collection, so this is not a huge problem (FIXME: this must be implemented as part of DC.) The third document will be needed to implement the data collection really, so that is also ok.

For the moment this is probably best described as a human readable document, something structured enough to get some information out, but not so fussy that people will get pissed off with it - in particular avoid XML!

Something like:

```
BEGIN PROJECT TS00
BEGIN CRYSTAL 12287

BEGIN AA_SEQUENCE
GIIYPGCFUIYGCUVGRVUTGRFTYUGFVETYFVYFDRYFDUUYFVYFRFVR
END AA_SEQUENCE

BEGIN HA_INFO
ATOM SE
NUMBER_PER_MONOMER 9
 - or -
NUMBER_TOTAL 9
END HA_INFO

BEGIN WAVELENGTH INFL   (*1)
WAVELENGTH 0.97950
F' -11.0
F'' 6.0
END WAVELENGTH INFL

BEGIN SWEEP INFL_DEF  (*2)
WAVELENGTH INFL
BEAM 109.0 105.0
IMAGE 12287_1_E1_001.img
DIRECTORY /Volumes/Arthur/JCSG Data/1vpj/data/blah/.../12287
END SWEEP INFL DEF

... etc ...

END CRYSTAL 12287
END PROJECT TS00
```

The records marked with `(*1)` and `(*2)` are user definable handles to make the linkup work. These will end up in the project/crystal/dataset hierarchy for MTZ files, or the equivalent therof in other data processing programs. They can also be used for defining file names and directories.

FIXME, need to add some use of directory structures to this system, so that the files are kept a little more organised. Would be a good exercise to put these files together for all of the TS data sets - that way I can make sure that they seem to make sense. Best to do it for all of the TS0X development examples as a dummy run.

Finally, there also needs to be scope in the file for comments. Any record beginning in "!" or "#" will be treated as such...

## 6.1 Derivation of Interesting Things

### 6.1.1 HA INFO

If there is a HA INFO element in there, then this suggests that this is a MAD data set and should be treated accordingly. If this is not present it is assumed that this will be a native data set, although the anomalous pairs will still be separated. This means:

- If there is no HA INFO, but F' etc. are present, an exception should be raised.
- If there is no HA INFO but multi wavelength data, likewise.
- If there is a HA INFO, but no F' etc., then use something like crosssec to generate them. This should be avoided near absorption edges.

If there is no sequence then it is assumed that the number of HA should be expressed as the total. If there is a sequence, then the number should be expressed per monomer or total - this can help when you have things like atoms on special positions, soaks etc. The number per monomer is likely if you have SeMet data.

If the ATOM record is specified, and it is SE, but the number per monomer is not defined and the AA SEQUENCE is, then just "count the M's" to get this number.

The F' numbers etc. should also be checked against the ATOM and WAVELENGTHS using crosssec, to see that the numbers are reasonable.

### 6.1.2 AA SEQUENCE

As described above, this will be used for deriving the number of HA if suitable circumstances occur. Otherwise, this will also be used for estimating the number of molecules in the ASU, the solvent content and obviously sequence docking.

### 6.2 Example: TS01

```
! F:\JCSG Data\1vr9>xia2find
! Sweep: F:\JCSG Data\1vr9\data\jcsg\als1\8.2.1\20050121\collection\TM0892\12847\12847_4_###
! Images: 1 to 180
! Detector class: adsc q210 2x2 binned
! Epoch from/to: 1106329906 1106332673
! Collected from: Fri Jan 21 17:51:46 2005
!             to: Fri Jan 21 18:37:53 2005
! Wavelength: 0.991870
! Distance: 150.000000
! Exposure time: 12.000000
! Beam: 105.099998 100.599998
! Oscillations: 161.000000 to 251.000000 (0.500000)
!
! Sweep: F:\JCSG Data\1vr9\data\jcsg\als1\8.2.1\20050121\collection\TM0892\12847\12847_5_###
! Images: 1 to 90
! Detector class: adsc q210 2x2 binned
! Epoch from/to: 1106332719 1106333435
! Collected from: Fri Jan 21 18:38:39 2005
!             to: Fri Jan 21 18:50:35 2005
! Wavelength: 0.991870
! Distance: 250.000000
! Exposure time: 5.000000
! Beam: 105.099998 100.599998
! Oscillations: 160.500000 to 250.500000 (1.000000)
!
! Sweep: F:\JCSG Data\1vr9\data\jcsg\als1\8.3.1\20050105\collect\TM0892\13140\13140_1_E1_###
! Images: 1 to 180
! Detector class: adsc q210 2x2 binned
! Epoch from/to: 1104976482 1104979591
! Collected from: Thu Jan 06 01:54:42 2005
!             to: Thu Jan 06 02:46:31 2005
! Wavelength: 0.979741
! Distance: 180.009995
! Exposure time: 3.001523
! Beam: 101.940002 101.050003
! Oscillations: 140.000000 to 230.000000 (0.500000)
!
! Sweep: F:\JCSG Data\1vr9\data\jcsg\als1\8.3.1\20050105\collect\TM0892\13140\13140_1_E2_###
! Images: 1 to 180
! Detector class: adsc q210 2x2 binned
! Epoch from/to: 1104976498 1104979607
! Collected from: Thu Jan 06 01:54:58 2005
!             to: Thu Jan 06 02:46:47 2005
! Wavelength: 1.019859
! Distance: 180.009995
! Exposure time: 3.005157
! Beam: 101.940002 101.050003
! Oscillations: 140.000000 to 230.000000 (0.500000)
```

```
BEGIN PROJECT TS01
BEGIN CRYSTAL 12847

BEGIN AA_SEQUENCE

MKVKKWVTQDFPMVEESATVRECLHRMRQYQTNECIVKDREGHFRGVVNKEDLLDLDLDSSVFNKVSLPD
FFVHEEDNITHALLLFLEHQEPYLPVVDEEMRLKGAVSLHDFLEALIEALAMDVPGIRFSVLLEDKPGEL
RKVVDALALSNINILSVITTRSGDGKREVLIKVDAVDEGTLIKLFESLGIKIESIEKEEGF

END AA_SEQUENCE

BEGIN WAVELENGTH NATIVE
WAVELENGTH 0.99187
END WAVELENGTH NATIVE

! high resolution native pass

BEGIN SWEEP NATIVE_HR
WAVELENGTH NATIVE
BEAM 109.0 105.0
IMAGE 12847_4_001.img
DIRECTORY F:\JCSG Data\1vr9\data\jcsg\als1\8.2.1\20050121\collection\TM0892\12847\
END SWEEP NATIVE_HR

! low resolution native pass

BEGIN SWEEP NATIVE_LR
WAVELENGTH NATIVE
BEAM 109.0 105.0
IMAGE 12847_5_001.img
DIRECTORY F:\JCSG Data\1vr9\data\jcsg\als1\8.2.1\20050121\collection\TM0892\12847\
END SWEEP NATIVE_LR

END CRYSTAL 12847

BEGIN CRYSTAL 13140

BEGIN AA_SEQUENCE

MKVKKWVTQDFPMVEESATVRECLHRMRQYQTNECIVKDREGHFRGVVNKEDLLDLDLDSSVFNKVSLPD
FFVHEEDNITHALLLFLEHQEPYLPVVDEEMRLKGAVSLHDFLEALIEALAMDVPGIRFSVLLEDKPGEL
RKVVDALALSNINILSVITTRSGDGKREVLIKVDAVDEGTLIKLFESLGIKIESIEKEEGF

END AA_SEQUENCE

BEGIN HA_INFO
ATOM SE
NUMBER_PER_MONOMER 5
```

```
END HA_INFO

BEGIN WAVELENGTH INFL
WAVELENGTH 0.979741
F' -10.0
F'' 3.2
END WAVELENGTH INFL

BEGIN WAVELENGTH LREM
WAVELENGTH 1.019859
F' -2.6
F'' 0.55
END WAVELENGTH LREM

BEGIN SWEEP INFL
WAVELENGTH INFL
BEAM 108.7 102.0
IMAGE 13140_1_E1_001.img
DIRECTORY F:\JCSG Data\1vr9\data\jcsg\als1\8.3.1\20050105\collect\TM0892\13140\
END SWEEP

BEGIN SWEEP LREM
WAVELENGTH LREM
BEAM 108.7 102.0
IMAGE 13140_1_E2_001.img
DIRECTORY F:\JCSG Data\1vr9\data\jcsg\als1\8.3.1\20050105\collect\TM0892\13140\
END SWEEP

END CRYSTAL 13140

END PROJECT TS01
```

## 6.3 Passing Things On

Once the data reduction has finished, the information about the relationship between data collection time and data set is lost. In particular, the *order* in which the data was collected is lost forever. To counteract this, the information should be added to the .xinfo file, so that later stages in the structure solution process will be able to make use of this information.

Much of the rest of the information can be represented by the MTZ file. However, a similar scheme will be needed when the data are output in something like scalepack unmerged format, shelx format &c.

### 6.4 Escalation

Now that I have a description of the input structure, the obvious thing to do is to map this into a set of classes in Python - XType classes (see next section.) This has the interesting outcome that this can practically *replace* my existing data structures, and can also provide a framework from which to hang all of the methods - an interesting turn of events.

## 7 XInfo, XType Classes

Once the .xinfo file was defined, for each stage in the xia2 pipeline (dc, dpa, ss) it becomes inevitable that this will begin to dominate the overall structure of the program, at least from the perspective of the data.

### 7.1 XSweep

The XSweep class, to represent the SWEEP record, contains the following methods:

- Getters & setters corresponding to Indexer.
- Getters & setters corresponding to Integrater.
- Scaler interface, too?

Scaling is managed at the XWavelength and XCrystal levels - although defining a resolution limit requires scaling, which in turn means that there has to be some kind of scaling interface at this level - though the options are not mutually exclusive.