

Diffraction Microscopy reconstruction software

November 25, 2008

Contents

1	Getting started	6
1.1	Updating this document	6
1.2	General remarks	6
1.3	Complete programs, and subroutine libraries	6
1.4	Subroutine library coding conventions	7
1.4.1	The file <code>dm.h</code>	7
1.4.2	<code>#defines</code> : system-level, and our own “invented” ones	8
1.4.3	Array memory allocation	9
1.4.4	Array ordering and centering conventions	9
1.4.5	Complex arrays	10
1.4.6	Using the predefined array structures	11
1.4.7	Testing your implementation of the subroutine libraries	13
2	Data files	14
2.1	Reading and writing ALS 9.0.1 <code>.nc</code> files	14
2.1.1	Routines for reading and writing <code>.nc</code> files in C/C++	15
2.1.2	Routines for reading and writing <code>.nc</code> files in IDL	15
2.2	HDF 5 files involved in iterative reconstructions	15
2.2.1	General structure of our HDF 5 files	15
2.2.2	The assembled diffraction intensity group <code>"/adi"</code>	16
2.2.3	Data error factor in the <code>"/adi"</code> group	17
2.2.4	Assembled diffraction sequence <code>"/ads"</code> group	17
2.2.5	Assembly info <code>"/ainfo"</code> group	17
2.2.6	Support mask <code>"/spt"</code> group	18
2.2.7	Iterate amplitude <code>"/itn"</code> group	18
2.3	3D data assembly <code>.3da</code> files	19
2.4	The subroutine library <code>dm_fileio.c</code>	19
2.4.1	Creating an <code>.h5</code> file and accessing an existing file	19
2.4.2	Reading and writing/updating the <code>"/comments"</code> group in C	19
2.4.3	Reading and writing the <code>"/adi"</code> group in C	22
2.4.4	Reading and writing the <code>"/comments"</code> group in C	23
2.4.5	Reading and writing the <code>"/ainfo"</code> group in C	23
2.4.6	Reading and writing the <code>"/spt"</code> group in C	25
2.4.7	Reading and writing the <code>"/itn"</code> group in C	26
2.5	IDL routines for file I/O	26
2.5.1	Reading and writing the <code>"/ainfo"</code> group in IDL	26

3	Procedures for diffraction data analysis	28
3.1	Cataloging files: the routine <code>dt_list_series.pro</code>	28
3.2	Merging individual ALS 9.0.1 exposures for one 2D view: Henry Chapman's merger	28
3.2.1	Using <code>merger_manager</code>	30
3.2.2	Wish list for enhancements to merger	31
3.3	Merging individual ALS 9.0.1 exposures for one 2D view: <code>commie</code>	31
3.3.1	A word on the <code>commie</code> distribution	31
3.3.2	Quick start: Assembly file	32
3.3.3	Quick start: Command line arguments (CLAs)	33
3.3.4	Quick start: Program flow	33
3.3.5	Further information	34
3.4	From 2D to 3D data: Anton Barty's program <code>Xewald</code>	34
3.4.1	Execution	34
3.4.2	Using	34
3.4.3	Output	35
3.4.4	Wish list for enhancements to <code>Xewald</code>	35
3.5	Ideas for programs we should have	35
3.5.1	<code>Cewald</code> : a C program for 3D data assembly	35
3.5.2	Estimating the center of a diffraction pattern	35
3.5.3	Estimating information content in a 2D view	36
3.5.4	Characterizing noise	36
3.5.5	Masking the beamstop and detecting saturated pixels	36
3.5.6	Displaying autocorrelation function from subregions of the diffraction data array	36
3.5.7	Estimating support from the autocorrelation	36
4	Iterative phasing programs	37
4.1	Pierre Thibault's <code>retriever</code>	37
4.2	Anton Barty's <code>r3d_mpi</code>	37
4.2.1	Wishlist for <code>r3d_mpi</code>	37
4.3	Wishlist for reconstruction programs	38
4.3.1	Application of Fourier modulus constraint with error factor per pixel	38
4.3.2	Application of support constraint	38
4.3.3	Calculation of the next iterate	38
4.3.4	Calculation $I_{\text{recon}}(f)/I_{\text{data}}(f)$ at a particular iterate	38
4.3.5	Calculation of error metric at a particular iterate	38
4.3.6	Script language for running iterations	38
4.3.7	Structure outside the support	38
5	Visualization of results	39
5.1	Routines for making movies of iterates	39
5.2	Routines for viewing 3D data	39
6	The array calculation subroutine library <code>dm_array.c</code>	40
6.1	MPI-enabled FFTs	41
6.1.1	Working with FFTW 2.1.5 MPI	41
6.1.2	Working with <code>dist_fft</code>	42
6.2	FFTs using <code>dm_array_fft()</code>	44
7	A roadmap for work at Stony Brook	45

A	Running MPI code	46
A.1	Running code using LAM MPI on Stony Brook's Apple cluster	46
A.1.1	Things to set up just once	46
A.1.2	Running your code	47
A.2	Running <code>dist_fft_test</code> using MPICH	47
A.2.1	Running MPICH code on a subset of the cluster	48
B	Running MPI DIST FFT	49
B.1	Compiler	49
B.2	Header file modification	49

List of Tables

List of Figures

1.1	Data and FFT centering of arrays	10
6.1	<code>dist_fft</code> 2D benchmark	43

Chapter 1

Getting started

1.1 Updating this document

This L^AT_EX document (`diffmic_recon.tex`) resides in the cvs directory `diffmic/doc/recon`. After you are finished with edits to this file, log in as `micros` on `xray1`, do `cd cvs/diffmic/doc/recon`, do a `cvs update`, and do `latex diffmic_recon` twice to make sure there are no errors. You can then simply run `make` to install the new version of this documentation on the `xray1` web page.

1.2 General remarks

This document describes software used for diffraction microscopy data acquisition and reconstruction. At present it has contributions from the X-ray Optics Group at the Department of Physics & Astronomy at Stony Brook University; from Anton Barty, Henry Chapman, and Stefano Marchesini at Lawrence Livermore National Laboratory; and from Pierre Thibault at Cornell University. The document you are reading has been generated from the file `diffmic_recon.tex` in the `diffmic/doc/recon` directory of the archive. This is available on-line as either a PDF file or a html link. An overview of the `diffmic` archive, CVS access to it, and so on is given in the document `diffmic/doc/diffmic.tex` which is available on-line as either a PDF file or a html link. This document includes a discussion of the general philosophy and goals of the `diffmic` archive.

1.3 Complete programs, and subroutine libraries

The CVS “diffmic” archive contains both complete programs, and subroutine libraries. Some of the complete programs pre-date the availability of the subroutine libraries, so naturally they don’t make use of the libraries. Nevertheless, it is hoped that the subroutine libraries described in Sec. 2.4 and Chapter 6 will form the basis for at least some of the future programming efforts for the following reasons:

- You can save yourself a lot of work by using already-tested routines for many things, rather than having to write everything yourself.
- The routines can be recompiled on various machines, so that a `main()` program written by a person with one type of system could be compiled and run by someone else on an entirely different system. These systems might include Windows laptops using the Cygwin Unix-like environment and FFTW3 for Fourier transforms, to Apple clusters using the MPI-enabled `dist_fft` library for Fourier transforms.
- Improvements made in various subroutines will immediately become available to all programs that use the subroutine with no effort beyond a simple recompilation. As an example, the subroutines recently gained MPI-compatibility which now automatically translates to any program that makes use of them.

This is the approach that has been used with great success by the protein crystallography community in the CCP4 project¹, where they state “The components of the whole system are thus a collection of programs using a standard software library to access standard format files (and a set of examples files and documentation) available for most Unix operating systems (including Linux), as well as Windows and Mac OS X.”

1.4 Subroutine library coding conventions

Having a common subroutine library means that a common set of programming conventions must be adhered to. As of this writing two libraries exist

- `dm_fileio.c` deals with file input-output from and to our custom defined file format based on the HDF5 standard. The library is defined in Sec. 2.4 and the file format in Sec. 2.2.
- `dm_array.c` provides a set of basic array manipulation routines. It is described in Ch. 6.

To be able to use these libraries your program must do the following

- initialize and finalize the subroutine libraries as follows

```
int my_rank,p;

/* Obtain info on number of processes and rank of current process.
 * Initialize MPI session if applicable.
 */
dm_init(&p,&my_rank);

/* call to library functions */
...

/* Finalize MPI session if applicable */
dm_exit();
```

Initialization will start an MPI-session if applicable and provide you with the number of processes (1 if you run in serial) and the rank of the current process. Exiting will finalize an existing MPI session if applicable.

- use the array structures defined in `dm.h`, see Sec. 1.4.1.
- use the macros for allocating memory as described in Sec. 1.4.3.
- use the types defined in `dm.h`, see Sec 1.4.1.
- be compiled using the `#defines` described in Sec. 1.4.2.

Finally, you can have a look at our test routines `dm_test_fileio.c` and `dm_test_array.c` in `diffmic/c/util/test` and the corresponding `Makefile` to see how it's done.

1.4.1 The file `dm.h`

In order to maximize transportability of code across different computers, the file `dm.h` in `diffmic/c/util` provides several definitions.

¹<http://www.ccp4.ac.uk/>

- It does `#include <sys/types.h>` so that one can refer to data types that (unlike `short int`, `int`, or `long int` which can vary in size from one compiler/processor to another) are of known size on any machine, like `u_int16_t` for a 16 bit unsigned integer, and so on.
- It also includes the file `endian.h` so that `BYTE_ORDER` is set to either `LITTLE_ENDIAN=1234` (such as on Intel processors) or `BIG_ENDIAN=4321` (such as on PowerPC and SPARC processors)².
- Depending on the presence or absence of the preprocessor `#define` of `DIST_FFT`, it either includes `dist_fft.h` or `fftw3.h` (see Sec. 6.2).
- It defines `dm_array_real` which is either `float` or `double`, according to the absence or presence of the preprocessor `#define` of `DM_ARRAY_DOUBLE` as described below.
- It defines `dm_array_index_t` which is an integer large enough to index an entire complex 3D array. This is presently set to `u_int32_t`. (Should this be set to `size_t`?)
- It defines array structures `dm_array_complex_struct`, `dm_array_real_struct`, and `dm_array_byte_struct`, as described in Sec. 1.4.6.
- For complex arrays, it defines the macros `c_re()` and `c_im()` to access real or imaginary elements as described in Sec. 1.4.5.
- For all array types it defines memory allocation macros that automatically allocate the right amount of memory whether we are running in parallel or serial, see Sec 1.4.3.
- It defines structures `dm_adi_struct` (see Sec. 2.2.2), `dm_spt_struct` (see Sec. 2.2.6), `dm_ainfo_struct` (see Sec. 2.2.5), and `dm_itn_struct` (see Sec. 2.2.7) containing parameters relevant to different data or calculations.
- It defines certain functions, in particular `dm_init` and `dm_exit` that are needed to correctly initialize the parameters `p` (as in number of processes) and `my_rank`. These parameters are passed along to subroutines to ensure compatibility whether we are using MPI or not. The source code of these functions is found in the file `dm.c`.

1.4.2 #defines: system-level, and our own “invented” ones

Several system-level `#defines` are worth noting, and are used when appropriate:

`__APPLE__` is set on Apple computers.

`__GNUC__` is set when using the GNU C compiler `gcc`.

`__CYGWIN__` is set in the Cygwin environment on Windows computers.

Others that we have “invented” for our subroutines are:

`__MPI__` signifies that the code should be compiled to use message passing interface capabilities on a parallel computer.

`DIST_FFT` signifies that the code should use the Apple `dist_fft` subroutines for Fourier transforms; otherwise the `FFTW3` subroutines are used by default. For more on this, see Sec. 6.2.

²Different computer types use different byte ordering schemes. In big endian byte order, the highest byte is stored first, so that `0x01020304` is stored in the order `0x01`, `0x02`, `0x03`, and `0x04`. Little endian byte order stores `0x01020304` as successive bytes `0x04`, `0x03`, `0x02`, and `0x01`. Big endian is the same as network byte order in the standard C routines `htons()` and `ntohs()` for 16 bit integers, and `htonl()` and `ntohl()` for 32 bit integers.

DM_ARRAY_DOUBLE means that the data types `dm_array_real` and `dm_array_complex` will be double rather than float. When using the `dist_fft` library make sure you also set the `DIST_FFT_USE_DOUBLE` flag to compile the `dist_fft` object files.

Others are used by the `dist_fft` library

`DIST_FFT_USE_DOUBLE` is used to tell the `dist_fft` routines to use double precision

`DIST_FFT_USE_INTERLEAVED_COMPLEX` is used to tell the `dist_fft` routines to use interleaved complex arrays as opposed to split complex arrays which is the default.

1.4.3 Array memory allocation

When programming portable code one has to be careful when allocating memory for arrays as the actual size of memory needed on each process depends on the actual number of processes (you can think of running the program in serial as having 1 process). Therefore the library provides memory allocation macros for each type of array defined in `dm.h` that will automatically determine the right amount of memory per process. In particular

- to allocate memory for a complex array use the construction

```
dm_array_complex *complex_array;
DM_ARRAY_COMPLEX_MALLOC (complex_array, npix, p);
...
DM_ARRAY_COMPLEX_FREE (complex_array);
```

where `npix` is the number of elements of the entire array and `p` is the number of processes returned by `dm_init()`.

- in analogy to allocate memory for a real array use the construction

```
dm_array_real *real_array;
DM_ARRAY_REAL_MALLOC (real_array, npix, p);
...
free (real_array);
```

- finally to allocate memory for a byte array use the construction

```
dm_array_byte *byte_array;
DM_ARRAY_BYTE_MALLOC (byte_array, npix, p);
...
free (byte_array);
```

1.4.4 Array ordering and centering conventions

The convention we assume for indexing arrays is that used in C language (“row-major” order). For a real array of dimensions $[n_x, n_y, n_z]$, array values are to be stored in “/adi” and “/spt” groups in the index order

$[0,0,0], [1,0,0], \dots, [(n_x-1),0,0], [0,1,0], \dots$

Therefore if one were to make a triple-indexed 3D array in C, it would be indexed as `array_3d[iz][iy][ix]`, but in fact most of the code uses a single array index of `array_3d[i]` which should be indexed as

```

for (iz=0; iz<local_nz; iz++)
  for (iy=0; iy<ny; iy++)
    for (ix=0; ix<nx; ix++)
      i = ix+iy*nx+iz*nx*ny;
      array_3d[i] = ...

```

The case of complex arrays such as used in the `"/itn"` group in HDF 5 files has a bit more to it, as described in Sec. 1.4.5.

Fast Fourier transform or FFT algorithms rearrange the data from input to output. It is important to agree on a convention for how the data is to be arranged; the convention we have adopted is as follows:

Data centering refers to having the center pixel at array location $[n_x/2, n_y/2, n_z/2]$. This is usually used in real or image space, and thus is the convention used for `"/spt"` and `"/itn"` groups.

FFT centering refers to having the zero frequency pixel (which would normally be in the center of a diffraction pattern) at array location $[0, 0, 0]$. This is usually used in reciprocal (or Fourier or diffraction) space, and thus is the convention used by the `"/adi"` group in files. The data must be arranged in this manner before routines such as `dm_fileio_write_adi` are called, and routines such as `dm_fileio_read_adi` will return an array that is FFT centered.

These conventions are illustrated in Fig. 1.1. The `.nc` raw data files violate this convention, in that they are recording data in reciprocal space but the CCD chip is meant to be positioned so that the center, zero-spatial-frequency pixel is at the array location $[n_x/2, n_y/2]$ for array indices starting from $[0, 0]$ (more on this in Sec. 3.5.2).

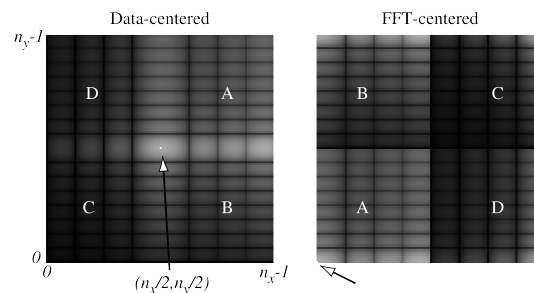


Figure 1.1: Schematic of the centering of arrays. Data in real space is meant to have the center of the image at pixel $[n_x/2, n_y/2]$, while data in reciprocal (or Fourier or diffraction) space has the zero spatial frequency pixel at array location $[0, 0]$.

1.4.5 Complex arrays

The story for arrays of complex numbers is, well, a bit more complex. There are two possible ways of arranging complex numbers (the indexing below is for data centering):

Interleaved: In this approach, the real and imaginary parts are stored in successive array locations in arrays of $[x, y, z]$:

Re[0,0,0], Im[0,0,0], Re[1,0,0], Im[1,0,0], ...

This is the most common approach.

Split: In this approach, the entire real array part is stored contiguously, followed by the entire imaginary part in arrays of $[x, y, z]$:

Re[0,0,0], Re[1,0,0], Re[2,0,0], ...,
Im[0,0,0], Im[1,0,0], Im[2,0,0], ...

This approach turns out to be slightly faster on some parallel computers (such as the `dist_fft` routines on an Apple cluster; see Sec. 6.2).

To make code that will work in either of these two conventions, you must follow some special mechanisms to work with complex arrays:

- The type `dm_array_index_t` is presently set to a 32 bit unsigned integer which works for arrays up to 1024^3 , but by using `dm_array_index_t` you allow for code to be modified to work with larger arrays and 64 bit indexing in the future by a simple re-definition of the typedef. Also, note that `sizeof(dm_array_complex)` will not necessarily give you the proper array size for storage because for split data it gives you the length of two memory references; see Sec. 1.4.3 on how to allocate memory for complex arrays.
- To manipulate a pixel in the complex array, do *not* refer to it as

```
real_part = *(complex_array+2*(ix+iy*nx+iz*nx*ny));
imaginary_part = *(complex_array+2*(ix+iy*nx+iz*nx*ny)+1);
```

but instead use the macros `c_re` and `c_im` for access:

```
real_part = c_re(complex_array, (ix+iy*nx+iz*nx*ny));
imaginary_part = c_im(complex_array, (ix+iy*nx+iz*nx*ny));
```

Also note that the typedefs of `dm_array_real` and `dm_array_complex` change between float and double according to the presence or absence of the #define variable `DM_ARRAY_DOUBLE`.

1.4.6 Using the predefined array structures

The `dm_array` library is written to make use of structures that incorporate arrays along with information on their dimensions. You should work with these array structures in a particular way. Here are some examples: for complex arrays, you should do

```
dm_array_complex_struct array_3d_cas;
dm_array_index_t ipix;
dm_array_real this_re, this_im;
int my_rank, p;

/* Initialize the session */
dm_init(&p, &my_rank)

array_3d_cas.nx = 512; /* for example */
array_3d_cas.ny = 512;
array_3d_cas.nz = 512;
array_3d_cas.npix = array_3d_cas.nx*array_3d_cas.ny*array_3d_cas.nz;

/* allocate memory */
DM_ARRAY_COMPLEX_MALLOC(array_3d_cas.complex_array, array_3d_cas.npix, p);

/* create a FFT-plan and store it with the array */
dm_array_fft(&array_2d_cas, p, DM_ARRAY_CREATE_FFT_PLAN,
            my_rank);

/* Get the real and imaginary part of the first pixel on each process */
ipix = 0;
this_re = c_re(array_3d_cas.complex_array, ipix);
this_im = c_im(array_3d_cas.complex_array, ipix);

/* Free up the memory allocated */
```

```

dm_array_fft (&array_2d_cas,p,DM_ARRAY_DESTROY_FFT_PLAN,
              my_rank);
DM_ARRAY_COMPLEX_FREE (complex_array);

/* Finish the session */
dm_exit ();

```

For real arrays, you should do

```

dm_array_real_struct real_3d_ras;
dm_array_index_t ipix;
dm_array_real this_real;
int my_rank,p;

/* Initialize the session */
dm_init (&p, &my_rank)

real_3d_ras.nx = 256; /* For example */
real_3d_ras.ny = 256;
real_3d_ras.nz = 256;
real_3d_ras.npix = real_3d_ras.nx*real_3d_ras.ny*real_3d_ras.nz;
real_3d_ras.real_array = DM_ARRAY_REAL_MALLOC (real_3d_ras.real_array, real_3d_ras.npix,p);

/* Get the value of the first pixel */
ipix = 0;
this_real = *(real_3d_ras.real_array+ipix);

/* Free up the array */
free (real_3d_ras.real_array);

/* Finish the session */
dm_exit ();

```

For byte arrays, you should do

```

dm_array_byte_struct byte_3d_bas;
dm_array_index_t ipix;
u_int8_t this_byte;
int my_rank,p;

/* Initialize the session */
dm_init (&p, &my_rank)

byte_3d_bas.nx = 256; /* For example */
byte_3d_bas.ny = 256;
byte_3d_bas.nz = 256;
byte_3d_bas.npix = byte_3d_bas.nx*byte_3d_bas.ny*byte_3d_bas.nz;
byte_3d_bas.byte_array = DM_ARRAY_BYTE_MALLOC (byte_3d_bas.byte_array,
                                                byte_3d_bas.npix,p);

/* Get the value of the first pixel */
ipix = 0;
this_byte = *(byte_3d_bas.byte_array+ipix);

```

```
/* Free up the array */
free(byte_3d_bas.byte_array);

/* Finalize the session */
dm_exit();
```

This way you only have to pass these structures by reference in the `dm_array` routines, such as in `dm_array_magnitude_complex(&real_3d_ras, &array_3d_cas, p);` which loads the array in `real_3d_ras` with the pixel-by-pixel magnitude of the array `array_3d_cas`.

1.4.7 Testing your implementation of the subroutine libraries

The subroutine libraries have been tested for several OSs and other parameters. If you are sticking to the above described programming conventions your code should pass the following tests

- using double or single precision using `fftw`
- using double or single precision using `dist_fft`
- using split or interleaved complex arrays when using `dist_fft`
- running in parallel or serial (right now this is equivalent to using `dist_fft` or `fftw`)

Please refer to the coding conventions in Sec. 1.4.2 to infer the correct `#defines` for compiling.

Chapter 2

Data files

In this section, we describe the following file formats:

Raw data files (.nc): With the Stony Brook apparatus at ALS 9.0.1, we are writing raw data files using the NetCDF format as described in Sec. 2.1. Oftentimes one will combine many of these files together (use background subtraction, *etc.* using Henry Chapman's `merger` program as described in Sec. 3.2;

Files used in iterative reconstructions: HDF 5 files with groups for the assembled diffraction intensity `"/adi"`, support mask `"/spt"`, and complex iterate `"/itn"` files. These files are described in Sec. 2.2, and C routines for reading and writing them are contained in the subroutine library `dm_fileio.c` (see Sec. 2.4).

2.1 Reading and writing ALS 9.0.1 .nc files

For historical reasons, the experimental endstation at the ALS saves its data as NetCDF¹ data files. Using subroutines available for just about any computer architecture, these binary files include self-documentation of the header variables, and cross-platform compatibility. NetCDF was a good choice some years ago; the latest proposed version (NetCDF-4) in fact uses HDF 5 (the format used for iterative reconstruction files as described in Sec. 2.2) for storing its data.

A specific set of parameters are intended to be updated and saved with each data file. These parameters are described in the C/C++ structure in `dt_par.h` which is in CVS `diffmic/c/expt`. Besides holding a variety of parameters and text strings, the structure also includes the starting array locations for each of four arrays per image:

`u_int16_pixel_data_array:`

There are `dt_par.u_int16_data_per_pixel` measurements of unsigned 16 bit integers at each pixel.

This is typically used to store raw CCD image data (`dt_par.datatype=DT_PAR_CCDIMAGE`), in which case one usually has `dt_par.u_int16_data_per_pixel=1`. When

`dt_par.datatype=DT_PAR_SCANNEDIMAGE`, `dt_par.u_int16_data_per_pixel=0` is the usual situation.

`u_int16_row_data_array:`

There are `dt_par.u_int16_data_per_row` unsigned 16 bit integers at each row. Usually zero.

`float_pixel_data_array:`

There are `dt_par.float_data_per_pixel` measurements of floating point values at each pixel. When `dt_par.datatype=DT_PAR_CCDIMAGE`, `dt_par.float_data_per_pixel=0` is typical. For `dt_par.datatype=DT_PAR_SCANNEDIMAGE`, the first array of `dt_par.n_cols×dt_par.n_rows` pixels makes up an image of the X positions in meters at points in the scan, while the part of this array represents the voltages sampled at each of those X positions. One can then map the image data onto a square grid in IDL (see the `read_dt.pro` procedure described in Sec. 2.1.2).

¹<http://www.unidata.ucar.edu/packages/netcdf>

`float_row_data_array:`

For `dt_par.datatype=DT_PAR_SCANNEDIMAGE`, this array contains the Y positions in meters at points in the scan.

2.1.1 Routines for reading and writing .nc files in C/C++

At the present time, the only routines providing access to these files are those in the C++ class `dt_image.cpp` in CVS `diffmic/c/util`, and by the IDL routines `read_dt.pro` and `write_dt.pro` in CVS `diffmic/idl`.

2.1.2 Routines for reading and writing .nc files in IDL

To read data files in IDL, use the `read_dt.pro` routine. It will return all the raw data arrays to you, but it will also calculate a single image in the BSIF common variable `image_data`. If you type `IDL> read_dt` by itself, you will see the routine's arguments. It returns to you the raw data arrays of integer and floating point pixel and row data, as well as loading the special array `image_data` that is defined in the common block of `bsif_common.pro`. Scanned images are handled specially; with them, the pre-scan and post-scan columns are stripped off and the image is resampled onto a regular grid in the fast axis direction. That is, `read_dt.pro` will return to you the raw data as saved by the scanning routines and the elements of the `dt_par` structure unmodified; at the same time, it will give you a modified version of `image_data`, `n_cols`, `x_dist`, `x_start`, and `x_stop` that is better for display of the image.

The routine also includes optional keywords for `psfile='myfile.ps'` to save a PostScript file that one can then print out, and options for `/display` to show the image on the screen and (for scanned images) the option `/click` to allow you to click on the image and see the position where you clicked.

2.2 HDF 5 files involved in iterative reconstructions

Iterative reconstructions involve several types of data. We want files containing these data to be transportable across machine architectures, and to be easily readable by MPI-enabled parallel computers. For these and other reasons, we have chosen to use the Hierarchical Data Format version 5 (HDF 5²) as our basic file format. Routines to work with these files are provided in the C subroutine library `dm_fileio.c` in CVS `diffmic/c/util` as described in Sec. 2.4, and the IDL routines described in Sec. 2.5. Examples of using these routines are shown in the files `dm_test_fileio.c` and `dm_test_fileio.pro` in the respective `test` directories. You can also view the contents of an HDF 5 file using a variety of other programs, including `h5dump` (a utility program provided as part of the HDF 5 distribution) and a Java viewer (`hdfview`, available at <http://hdf.ncsa.uiuc.edu/hdf-java-html/hdfview/>).

2.2.1 General structure of our HDF 5 files

The general structure we will use for our HDF 5 files is as follows:

- The group `"/comments"`, which should be part of every `.h5` file we make! These comments are meant to track changes to/operations done on the data. It is intended that each program that modifies the data will add a short text string or strings describing the modification done. You can view these comments with the program `dm_fileinfo` which is in CVS `diffmic/c/util`. Within the `comments` group you will find:

`comment_string_length` tells the maximum length of each string. (We use fixed string lengths because HDF 5 variable length strings are not supported by IDL 6.2).

`n_comment_strings` tells how many strings are in the array.

`specimen_name` is a string for a short description of the specimen.

²<http://hdf.ncsa.uiuc.edu/HDF5/>

`collection_date` is a string that is meant to hold the `C systime()` string from when the original data was recorded.

`comment_strings` is the array of strings, each string of which is no longer than `comment_string_length-1` characters in length.

- The group `"/adi"`, for those files that contain assembled diffraction intensities (Sec. 2.2.2).

The attribute `adi_version` is used to indicate to the `dm_fileio` routines any changes in the structure of information in the `"/adi"` group.

`adi_array` contains the actual data.

`adi_struct` contains a structure of parameters pertaining to the data.

`adi_error_array` is an optional array as described in Sec. 2.2.3.

- The group `"/ainfo"`, that stores important information about the files that were used for the assembly of an `"/adi"` file (Sec. 2.2.5).

- The group `"/spt"`, for those files that contain support masks (Sec. 2.2.6).

The attribute `spt_version` is used to indicate to the `dm_fileio` routines any changes in the structure of information in the `"/spt"` group.

`spt_array` contains the actual mask. This is usually a byte image.

`spt_struct` contains a structure of parameters pertaining to the mask.

- The group `"/itn"`, for those files that contain complex iterates (Sec. 2.2.7).

The attribute `itn_version` is used to indicate to the `dm_fileio` routines any changes in the structure of information in the `"/itn"` group.

`itn_array` contains the actual complex iterate. This array will have C array ordering `[nz,ny, nx,2]` so that the index between interleaved real and imaginary parts will vary most rapidly.

`itn_struct` contains a structure of parameters pertaining to the mask.

Floating point data arrays will be written according to the value of `typedef dm_array_real`: that is, as double-precision if `DM_ARRAY_DOUBLE` is specified at compile time (C) or as a keyword (IDL), or otherwise as a single-precision float. When reading the file, the file's native data type will be converted into float or double based on the type of `dm_array_real`. Our HDF 5 routines automatically handle any byte-swapping that might be required.

2.2.2 The assembled diffraction intensity group `"/adi"`

The raw data `.nc` files described in Sec. 2.1 contain considerable detail on the experimental settings used to record a particular exposure. However, the goal in the experiment is to produce a single, high quality file representing the processed diffraction data (possibly along with some indication of the error of various pixels). The `"/adi"` group contains both the data, and a structure `adi_struct` of related parameters. The elements of this structure are as follows:

`photon_scaling`: multiply `adi_array` by this factor to convert it into photons per pixel. Set to zero if unknown.

`error_scaling`: scaling factor s_e to apply to `error_array`. See 2.2.3.

`lambda_meters`: x-ray wavelength in meters.

`camera_z_meters`: CCD camera distance from sample.

`camera_x_pixelsize_meters`: size of camera pixels in the `nx` direction, in meters.

camera_y_pixelsize_meters: size of camera pixels in the n_x direction, in meters.

theta_x_radians: GMR position (rotation about the horizontal transverse direction) to which the sample was set to be at. *Obsolete since already defined in "/ainfo" group?*

theta_y_radians: orientation of the specimen about the vertical transverse direction. Set to zero if unknown. *Obsolete since already defined in "/ainfo" group?*

theta_z_radians: orientation of the specimen about the beam axis direction. Set to zero if unknown. *Obsolete since already defined in "/ainfo" group?*

xcenter_offset_pixels: location of the true center of the array relative to pixel $n_x/2$.

ycenter_offset_pixels: location of the true center of the array relative to pixel $n_y/2$.

2.2.3 Data error factor in the "/adi" group

The optional array `adi_error_array` ($e_{i,j,k}$) can be used to store information on the error of each voxel in an assembled data file. When multiplied by `error_scaling` (or $s_e \cdot e_{i,j,k}$), this gives a number which assigns an error for each diffraction space pixel as a multiple of the square root of its intensity. That is, if $s_e \cdot e_{i,j,k} = 1$, then it will be assumed that the 1σ root variance of the measurement at that pixel is equal to the square root of the number of photons. Voxels with $s_e \cdot e_{i,j,k} > 1$ will have greater than the usual error (and thus their intensities will be less vigorously enforced when applying the Fourier modulus constraint), while voxels with $s_e \cdot e_{i,j,k} = 0$ will assumed to be unknown so that no constraint will be placed on their Fourier modulus (for example, in the case of pixels behind the beamstop where no information is known).

2.2.4 Assembled diffraction sequence "/ads" group

Section to be written.

2.2.5 Assembly info "/ainfo" group

Since most of our raw data is assembled into either complete 2D diffraction images or 3D datacubes, it is important to keep track of all important experimental settings that were in effect when the raw data files were recorded. Thus, the "/ainfo" group is comparable to the `adi_struct` of the "/adi" group, with the difference that all tags are initialized as arrays that contain the respective values of every single file that has been used for the assembly. Due to that the `adi_struct` becomes partly obsolete and is only being used for tags that are likely to have the same value for all files (e.g. `lambda_meters`). The "/ainfo" group as it is written to the HDF5 file consists of the following tags (Note that there might be additional parameters that are only being used within the I/O routines and that might differ depending on the implementation):

The attribute `ainfo_version` is used to indicate to the `dm_fileio` routines any changes in the structure of information in the `ainfo_group`.

`file_directory` contains the path to the folder where all the files that were used for the assembly are stored. *Obsolete since we are giving full filenames in `filename_array`?*

`filename_array` contains an array of filenames of all the files used for the assembly.

`system_time_array` stores the creation date and time for every file that was used for the assembly.

`camera_x_pixelsize_m_array` stores the size of camera pixels in meters for every file used for the assembly. *Obsolete since already defined in "/adi" group?*

`camera_y_pixelsize_m_array`: see above. *Obsolete since already defined in "/adi" group?*

`theta_x_radians_array`: GMR position for each file .

`xcenter_array`: location of the center of the array.

`ycenter_array`: see above. (*Maybe change to `ycenter_offset_pixels`?*)

`n_frames` stores the number of files that were used for the assembly. This value is entered by the user and has to be checked carefully since it determines the size of the above arrays.

`string_length`: the minimum string length required to store the longest filename, systime, or file directory. Automatically determined in the IDL implementation. For C the user has to define a keyword.

For information about how to read and store information from/to the `"/ainfo"` group, see Sec. 2.4.5 for the C implementation and Sec. 2.5.1 for the IDL implementation.

2.2.6 Support mask `"/spt"` group

The support mask (a real-space array of 1 byte per pixel) defines a support constraint in 2D or 3D. The parameters contained in the associated `spt_struct` are:

`support_scaling` (s_s): multiply `spt_array` by this factor (or $s_s \cdot s_{i,j,k}$) to give a value which is 0 for fully outside the support, and 1 for fully inside the support. This allows one to use, for example a gray-scale byte support array by setting $s_s = 1/255$.

`pix_x_meters`: real space pixel size in the first dimension.

`pix_y_meters`: real space pixel size in the second dimension.

`pix_z_meters`: real space pixel size in the third dimension. The value will be ignored if the mask is 2D.

2.2.7 Iterate amplitude `"/itn"` group

At the end of an iterative calculation, we wish to save the resulting complex amplitude in real space as `itn_array`. It may on occasion be desirable to save intermediate calculation results as well. The `itn_struct` information associated with these files is:

`pix_x_meters`: real space pixel size in the first dimension.

`pix_y_meters`: real space pixel size in the second dimension.

`pix_z_meters`: real space pixel size in the third dimension. This value must be present but is ignored for 2D arrays.

- `photon_scaling` (s_p): multiply `itn_array` by this factor to convert it into photons per pixel. Set to zero if unknown.
- `iterate_count` (i ; `uint32`): counter of iteration number.

2.3 3D data assembly `.3da` files

To assemble 2D `"/adi"` diffraction intensity files into a 3D `"/adi"` file, it may be useful to have a simple ASCII text file format called `.3da` which would have lines:

```
filename,theta_x,theta_y,shift_x,shift_y,i_scaling  
filename,theta_x,theta_y,shift_x,shift_y,i_scaling  
...
```

The parameters `theta_x` and `theta_y` give the present guess of the 2D diffraction pattern's orientation in degrees. The parameters `shift_x` and `shift_y` give the shift (in pixels) of the center of each 2D `"/adi"` file relative to the usual position as defined in Sec. 1.4.4. The parameter `i_scaling` gives a multiplying factor for each file to scale its intensities to the 3D set's average.

It could be worthwhile for Xewald (Sec. 3.4) to be able to write out `.3da` files, and for Cewald (Sec. 3.5.1) to be able to read and write these files.

2.4 The subroutine library `dm_fileio.c`

The subroutine library `dm_fileio.c` in CVS `diffmic/c/util` provides a set of routines for reading and writing the above file types.

A sample routine that illustrates the usage of our functions can be found in `diffmic/c/util/test_js_copy`. Jan to write more. Note that there's a routine in here called `dm_write_pngfile()` that will write a 2D byte array image out to a `.png` graphics file. This can be used to allow an iterative reconstruction program to write out a snapshot of the reconstruction from time to time so that we can see how the reconstruction is proceeding.

Updating `dm_fileio` It is desirable that `dm_fileio` is updated by as few people as possible assuring that these updates are compatible with the rest of the file format utility routines and the IDL versions. I suggest that, in case of doubt, any requested add-ons are sent to Jan Steinbrener, who will try to implement them in a timely manner. Additionally there is a textfile called `dm_fileio_changelog` where any changes should be documented in detail and don't forget to add your initials in case I need to get back to you about it.

2.4.1 Creating an `.h5` file and accessing an existing file

The following functions are provided to create new or access existing `.h5` files

- `int dm_h5_create(char *filename, hid_t *ptr_h5_file_id, char *error_string)` As its name suggests this function will create a new `.h5` file at the location specified with `*filename`. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.
- `int dm_h5_openwrite(char *filename, hid_t *ptr_h5_file_id, char *error_string)` This function will open an existing `.h5` file specified by `*filename` with write access. Use this if you wish to modify the contents of the file. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.
- `int dm_h5_openread(char *filename, hid_t *ptr_h5_file_id, char *error_string)` This function will open an existing `.h5` file specified by `*filename` with read-only access. Use this if you just wish to examine the contents of the file. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.

After you're done with the file you should release its identifier by calling `void dm_h5_close(hid_t h5_file_id)`.

2.4.2 Reading and writing/updating the `"/comments"` group in C

To deal with the `"/comments"` group the library provides you with nine functions:

- `int dm_h5_create_comments(hid_t h5_file_id, dm_comment_struct *ptr_comment_struct, char *error_string)` This function is called when first adding a `"/comments"` group to an open `.h5` file. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.
- `int dm_h5_add_comments(hid_t h5_file_id, dm_comment_struct *ptr_comment_struct, char *error_string)` This function will merely add comments to an existing `"/comments"` group in an open `.h5` file. Note that it will not tamper with the group members `specimen_name` and `collection_date`. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.

- `int dm_h5_read_comments_info(hid_t h5_file_id, int *ptr_n_strings, int *ptr_string_length, char *error_string)` Use this function to determine how much memory you need to allocate to read out the actual comments. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.
- `int dm_h5_read_comments(hid_t h5_file_id, dm_comment_struct *ptr_comment_struct, char *error_string)` This function will read the comments, specimen name and collection date from an existing `"/comments"` group in an open `.h5` file. It will return either `DM_FILEIO_SUCCESS` or `DM_FILEIO_FAILURE`.
- `void dm_add_string_to_comments(char *string_to_add, dm_comment_struct *ptr_comment_struct)` Use this function to add a comment string to an already allocated `string_array` in the `comment_struct`. It will make sure that all strings are properly zero-terminated and aligned.
- `void dm_add_specimen_name_to_comments(char *string_to_add, dm_comment_struct *ptr_comment_struct)` Same as before except that it will add the specimen name to an already allocated `specimen_name` in the `comment_struct`.
- `void dm_add_collection_date_to_comments(char *string_to_add, dm_comment_struct *ptr_comment_struct)` Same as before except that it will add the collection date to an already allocated `collection_date` in the `comment_struct`.
- `void dm_clear_comments(dm_comment_struct *ptr_comment_struct)` This function makes sure that an already allocated `comment_struct` is filled up with null-bytes. This should be your first call after you allocated the `comment_struct`.
- `void dm_print_comments(FILE *fp_out, char *preceding_string, dm_comment_struct *ptr_comment_struct, char *trailing_string)` Use this function to display the contents of `comment_struct` or to save it to a file. You can specify the layout via the `preceding_string` and the `trailing_string` arguments.

Now let's look at some examples:

Preparing and writing a `comment_struct` to an opened `.h5` file First you need to define a maximum number of strings and stringlength, then you can allocate the `comment_struct`. Don't forget to call `dm_clear_comments` after you allocated memory.

```
my_comment_struct.n_strings_max = MAX_COMMENT_STRINGS;
my_comment_struct.string_length = STRLEN;
my_comment_struct.string_array =
    (char *)malloc(my_comment_struct.n_strings_max*
                  my_comment_struct.string_length);
my_comment_struct.specimen_name =
    (char *)malloc(my_comment_struct.string_length);
my_comment_struct.collection_date =
    (char *)malloc(my_comment_struct.string_length);
dm_clear_comments(&my_comment_struct);
```

Now you can add some strings and a specimen name and a collection date. Note that `dm_add_string_to_comments` will automatically increment the `n_strings` counter in `comment_struct`.

```
dm_add_string_to_comments("This is the first comment line.",
                          &my_comment_struct);
dm_add_string_to_comments("This is something equally useless.",
                          &my_comment_struct);
dm_add_string_to_comments("Now this is getting ridiculous!",
                          &my_comment_struct);
```

```

dm_add_string_to_comments("Let's get this over with already...",
    &my_comment_struct);
dm_add_specimen_name_to_comments("Yeast cell",&my_comment_struct);
time(&t);
dm_add_collection_date_to_comments(ctime(&t), &my_comment_struct);

```

Now we are ready to write the comments to an already opened .h5 file

```

if (dm_h5_create_comments(h5_file_id, &my_comment_struct,
    error_string) != DM_FILEIO_SUCCESS) {
    printf("%s\n", error_string);
    dm_h5_close(h5_file_id);
    exit(1);
}

```

Reading a comment_struct from an opened .h5 file First thing to do is to query the size of the comment string array and the stringlength so that we can allocate memory. Again, don't forget to call `dm_clear_comments`.

```

if (dm_h5_read_comments_info(h5_file_id, &n_strings, &string_length,
    error_string) != DM_FILEIO_SUCCESS) {
    printf("%s\n", error_string);
    dm_h5_close(h5_file_id);
    exit(1);
}

```

```

my_comment_struct.n_strings_max = n_strings;
my_comment_struct.string_length = string_length;

```

```

my_comment_struct.string_array =
    (char *)malloc(my_comment_struct.n_strings_max*
        my_comment_struct.string_length);
my_comment_struct.specimen_name =
    (char *)malloc(my_comment_struct.string_length);
my_comment_struct.collection_date =
    (char *)malloc(my_comment_struct.string_length);

```

```

dm_clear_comments(&my_comment_struct);

```

Now we can read the actual comments.

```

if (dm_h5_read_comments(h5_file_id, &my_comment_struct,
    error_string) != DM_FILEIO_SUCCESS) {
    printf("%s\n", error_string);
    dm_h5_close(h5_file_id);
    exit(1);
}

```

Updating a comment_struct in an opened .h5 file To add a new comment to an existing `comment_struct`, you want to perform the following steps. Note that this will not change the `specimen_name` nor the `collection_date`: First define the maximum number of strings and the stringlength then allocate memory and clear the `comment_struct`.

```

my_comment_struct.n_strings_max = MAX_COMMENT_STRINGS;
my_comment_struct.string_length = STRLEN;

```

```

my_comment_struct.string_array =

```

```

        (char *)malloc(my_comment_struct.n_strings_max*
            my_comment_struct.string_length);
my_comment_struct.specimen_name =
        (char *)malloc(my_comment_struct.string_length);
my_comment_struct.collection_date =
        (char *)malloc(my_comment_struct.string_length);

dm_clear_comments(&my_comment_struct);

```

Then add your comments first to the `comment_struct` and then to the file by calling

```

dm_add_string_to_comments("In a world without",
    &my_comment_struct);
dm_add_string_to_comments("FENCES and WALLS, ",
    &my_comment_struct);
dm_add_string_to_comments("we don't need no WINDOWS or GATES!",
    &my_comment_struct);

if (dm_h5_add_comments(h5_file_id, &my_comment_struct,
    error_string) != DM_FILEIO_SUCCESS) {
    printf("%s\n", error_string);
    dm_h5_close(h5_file_id);
    exit(1);
}

```

2.4.3 Reading and writing the "/adi" group in C

First we'll provide an overview of all functions related to the "/adi" group:

`dm_h5.write_adi(h5_file_id, *ptr_adi_struct, *ptr_adi_array_struct, ptr_adi_error_array_struct, *error_string)`: Creates an "/adi" group within an already-opened HDF5 file. Copies information from an existing `dm_adi_struct` together with data arrays to this "/adi" group. If `adi_error_array.npix = 0` then no `adi_error_array` will be added to the "/adi" group.

`dm_h5.adi_group_exists(h5_file_id)`: Checks whether or not an "/adi" group is present within an opened HDF5 file.

`dm_h5.read_adi_info(h5_file_id, *ptr_nx, *ptr_ny, *ptr_nz, *ptr_error_is_present, *ptr_adi_struct, *error_string)`: reads the ADI structure and the size of the ADI array from an already-opened HDF5 file. It also determines if there is an `adi_error_array` in the file or not. If `adi_error_array` is not present, you should not allocate memory for it and you should set `nx = ny = nz = npix = 0` in any `adi_error_array_struct` that you might create.

`dm_h5.read_adi(h5_file_id, *ptr_adi_array_struct, *ptr_adi_error_array_struct, error_string)`: This routine reads in `adi_array_struct`, and `adi_error_array_struct`. If there is no `adi_error_array_struct` data to be read in, then `nx = ny = nz = npix = 0` is set in the structure. If `adi_error_array_struct.npix` is set to zero before calling this routine, then `adi_error_array` will not be read in even if it is present in the file.

`dm_h5.insert_adi_struct_members(datatype)`: This internal routine does the `H5Tinsert()` calls to build up the `adi_struct` variables.

Preparing and writing an `adi_struct` to an "/adi" group

Reading an `adi_struct` from an `"/adi"` group

2.4.4 Reading and writing the `"/comments"` group in C

Again we start with an overview over all relevant functions:

`dm_h5_write_comments(h5_file_id, *ptr_comment_struct, *error_string)`: Add comments to an already-opened HDF5 file.

`dm_h5_read_comments(h5_file_id, *ptr_comment_struct, *error_string)`: This routine reads in the comments from an already-opened HDF5 file.

`dm_clear_comments(*ptr_comment_struct)`: This routine clears the comments of the comments string array.

`dm_add_string_to_comments(*string_to_add, *ptr_comment_struct)`: This routine adds a string to the comment string array. If the length of the new string is beyond of what can be accommodated, the string to add is truncated.

`dm_add_specimen_name_to_comments(*string_to_add, *ptr_comment_struct)`: This routine updates `specimen_name` within the comments.

`dm_add_collection_date_to_comments(*string_to_add, *ptr_comment_struct)`: This routine updates `collection_date` within the comments.

`dm_print_comments(*fp_out, *preceding_string, *ptr_comment_struct, trailing_string)`: This routine prints out the string array line-by-line to whatever is specified in `fp_out`. The format can be modified with the `preceding_string` and `trailing_string` parameters.

Preparing and writing a `comment_struct` to a `"/comments"` group

Reading a `comment_struct` from a `"/comments"` group

2.4.5 Reading and writing the `"/ainfo"` group in C

Let's start with an overview of all functions related to the `"/ainfo"` group before we describe the interaction of these functions:

`dm_h5_write_ainfo(h5_file_id, *ptr_ainfo_struct, *error_string)`: Creates an `"/ainfo"` group within an open HDF5 file and writes an existing `dm_ainfo_struct` (defined in `dm.h`) to this `"/ainfo"` group.

`dm_h5_read_ainfo_info(h5_file_id, *ptr_n_names, *ptr_string_length, *error_string)`: This routine reads the parameters `n_frames` and `string_length` from an existing `"/ainfo"` group within an open HDF5 file. It thus provides information on how much memory we need to allocate for the arrays.

`dm_h5_read_ainfo(h5_file_id, *ptr_ainfo_struct, *error_string)`: Reads out the `"/ainfo"` group from an open HDF5 file and stores it into an existing `dm_ainfo_struct`.

`dm_clear_ainfo(*ptr_ainfo_struct)`: Clears all arrays and parameters contained in an existing `dm_ainfo_struct`. Should be called whenever a new `dm_ainfo_struct` is initialized.

`dm_add_file_directory_to_ainfo(*directory_to_add, *ptr_ainfo_struct)`: stores a file directory in an existing `dm_ainfo_struct`.

`dm_add_filename_to_ainfo(*filename_to_add, *ptr_ainfo_struct)`: adds a filename to an existing `dm_ainfo_struct`.

`dm_add_systime_to_ainfo(*systime_to_add, *ptr_ainfo_struct)`: adds a systime to an existing `dm_ainfo_struct`.

`dm_add_double_to_ainfo(*tagname, *double_to_add, *ptr_ainfo_struct, *error_string)`: adds a double value to one of the double arrays within an existing `dm_ainfo_struct`. The array is specified by `*tagname`.

`dm_read_ainfo_from_csv(*csv_filename, *ptr_ainfo_struct, *error_string)`: this function reads in a specially formatted textfile, containing tagnames together with csv-separated values and stores these in the respective arrays of an existing `dm_ainfo_struct`. For guidelines on how to create such a text file see below (or click 2.4.5).

`dm_check_ainfo(*ptr_ainfo_struct, *error_string)`: checks the size of the arrays of an existing `dm_ainfo_struct` before it is written to the HDF5 file. This routine is **automatically** called by `dm_h5_write_ainfo`.

`dm_print_ainfo(*fp_out, *preceding_string, *ptr_ainfo_struct, *trailing_string)`: prints out an existing `dm_ainfo_struct` to `*fp_out`. The format can be edited using the preceding and trailing string parameters.

The principal read and write functions for the `"/ainfo"` group that access and store data on disk are `dm_h5_write_ainfo`, `dm_h5_read_ainfo_info`, and `dm_h5_read_ainfo`. In order for those functions to work properly it is important that the `dm_ainfo_struct` is initialized correctly. Various other functions are defined to correctly store whole arrays or single values within the `dm_ainfo_struct`. Again, an example of how to deal with those functions is given in `/diffmic/c/util/test`. The following paragraphs illustrate typical read/write procedures.

Preparing and writing `dm_ainfo_struct` to an `"/ainfo"` group

Initialize an instance of `dm_ainfo_struct` which is defined in `dm.h`.

Define `dm_ainfo_struct.string_length` and `dm_ainfo_struct.n_frames_max`. Use this to allocate memory for the char- and double arrays. You also need to define `dm_ainfo_struct.ainfo_tags` if you plan on using `dm_read_ainfo_from_csv`. It tells the function the number of tags it has to look for in the text file.

Use `dm_clear_ainfo` to finish initialization of `dm_ainfo_struct`.

Then store the actual number of files used for the assembly into `dm_ainfo_struct.n_frames`. This tag is particularly important since it is later-on used in several functions to check the size of the arrays. Make sure to enter the correct value here!

Now store values into the arrays. There are several ways to do so. The functions provided (see above) offer the possibility to copy information into the arrays value by value by adding a single parameter at a time, or to read the whole `dm_ainfo_struct` from a specially formatted text-file (see below). It is possible to leave some arrays un-initialized or only partly filled. In that case the code will simply add default values (which should make clear that the user did not store anything) for the missing values.

Use `dm_h5_write_ainfo` to create an `"/ainfo"` group within an open HDF5 file and to write the `dm_ainfo_struct` to this `"/ainfo"` group.

Reading `dm_ainfo_struct` from an `"/ainfo"` group

Use `dm_read_ainfo_info` to determine the size of the arrays. It will return `dm_ainfo_struct.string_length` and a `dm_ainfo_struct.n_frames`. Note that you'll also need to set a value for `n_frames_max`, as this is used to check whether your computer can handle large arrays. Usually it should be fine to just set it to `n_frames` that you just determined from `dm_read_ainfo_info`.

Initialize an `dm_ainfo_struct` using the parameters we just determined.

Use `dm_h5_read_ainfo` to read the arrays from an open HDF5 file into the `dm_ainfo_struct`.

You can use `dm_print_ainfo` to create a print-out of `dm_ainfo_struct`.

Format specifications for the csv-file If you use `dm_read_ainfo_from_csv` make sure that the text file obeys to the following formatting rules: The general syntax is:

```
tag=value1, value2, value3, ..., valueN
```

Different tags are separated by a newline character (*i.e.* every tag and its associated values occupy exactly one line). The tag specifies the array to which its values belong to. Possible tags are (more can be added upon request. Case sensitive!!):

```
directory
filenames
systimes
camera_x
camera_y
gmr_x
theta_x (Note: gmr_x is converted to theta_x which is in radians)
xcenter
ycenter
```

Note: If the text file contains more tags than specified in `dm_ainfo_struct.ainfo_tags` the code will ignore the extra tags.

2.4.6 Reading and writing the `"/spt"` group in C

Again, we start with an overview of relevant functions:

`dm_h5_write_spt(h5_file_id, *ptr_spt_struct, *ptr_spt_array_struct, *error_string):` Add a support mask to an already-opened HDF5 file.

`dm_h5_spt_group_exists(h5_file_id):` This routine determines if an `"/spt"` group exists.

`dm_h5_read_spt_info(h5_file_id, *ptr_nx, *ptr_ny, *ptr_nz, *ptr_spt_struct, error_string):` This routine reads the `spt_struct` and the size of the SPT array from an already-opened HDF5 file.

`dm_h5_read_spt(h5_file_id, *ptr_spt_array_struct, *error_string):` This routine reads in `spt_array_struct`.

`dm_h5_insert_spt_struct_members(datatype):` This internal routine does the `H5Tinsert()` calls to build up the `spt_struct` variables

Preparing and writing a `spt_struct` to a `"/spt"` group

Reading a `spt_struct` from a `"/spt"` group

2.4.7 Reading and writing the `"/itn"` group in C

Again, we start with an overview of relevant functions:

`dm_h5_write_itn(h5_file_id, *ptr_itn_struct, *ptr_itn_array_struct, *error_string):` Add a complex iterate to an already-opened HDF5 file.

`dm_h5_itn_group_exists(h5_file_id):` This routine determines if an `"/itn"` group exists.

`dm_h5_read_itn_info(h5_file_id, *ptr_nx, *ptr_ny, *ptr_nz, *ptr_itn_struct, error_string):` This routine reads the `itn_struct` and the size of the ITN array from an already-opened HDF5 file.

`dm_h5_read_itn(h5_file_id, *ptr_itn_array_struct, *error_string):` This routine reads in `itn_array_struct`.

`dm_h5_insert_itn_struct_members(datatype):` This internal routine does the `H5Tinsert()` calls to build up the `itn_struct` variables

Preparing and writing an `itn_struct` to an `"/itn"` group

Reading an `itn_struct` from an `"/itn"` group

2.5 IDL routines for file I/O

IDL routines to do the same jobs as the C library `dm_fileio.c` are contained in CVS `difffmic/idl/util`.

2.5.1 Reading and writing the `"/ainfo"` group in IDL

The IDL implementation of these functions comes off a little bit lighter than the C implementation. These are the main functions:

`dm_init_ainfo_struct, ainfo_struct, ainfo_member_names:` Creates an instance of `dm_ainfo_struct` and also an array of lower-case member names for use by `H5T_idl_create`.

`dm_ptr_free_ainfo, ainfo_struct:` Checks if pointers in `dm_ainfo_struct` are initialized and frees them if so. Use this at the end of your routines to clear up any pointers that might have been casted within `dm_ainfo_struct`.

`dm_h5_write_ainfo, h5_file_id, ainfo_struct, error_string, ainfo_group=ainfo_group:` same function as the C version. Additionally, it offers the possibility to define the keyword `ainfo_group`. Set this keyword if you want to modify an existing `ainfo`-structure. Also checks whether all arrays have the same dimensionality and adds default values if needed. IDL particularity: No need to define a string length since this function determines the length of the longest string in all string arrays and stores it in `dm_ainfo_struct.string_length`.

`dm_h5_read_ainfo, h5_file_id, ainfo_struct, error_string:` Same as its C counterpart except that it initializes an `dm_ainfo_struct` by itself.

`dm_read_ainfo_from_csv, csv_filename, dm_ainfo_struct, error_string:` Same functionality as C counterpart. For details on how the text file should be formatted see 2.4.5.

`dm_add_ainfo, filename, csv_filename, n_frames`: This is a work-around solution until `merger_manager` is capable of writing its own working HDF5 files. Until then use `merger` to produce the HDF5 file and insert any ainfo-information afterwards by a call to this routine.

Since it is possible and easy to dynamically resize arrays in IDL we don't need functions to add values to the arrays. The only function provided for that which reads the whole `dm_ainfo_struct` from a specially formatted textfile. An example of how to perform typical operations with that library is given in `/diffmic/idl/util/test`. The following paragraphs illustrate typical read/write procedures.

Preparing and writing `dm_ainfo_struct` to an `"/ainfo"` group

Initialize an instance of `dm_ainfo_struct` by calling `dm_init_ainfo_struct`.

Associate the arrays with their respective values, either manually or use `dm_read_ainfo_from_csv` to read in the whole thing at once.

Determine and define `dm_ainfo_struct.n_frames`. Like for the C implementation, this is a crucial parameter since it is being used to determine if the arrays have the correct length or if they need to be filled up. You don't need to define a `dm_ainfo_struct.string_length` since this is determined automatically when calling `dm_h5_write_ainfo`.

Call `dm_h5_write_ainfo` to write the existing `dm_ainfo_struct` to an `"/ainfo"` group within an open HDF5 file.

At the very end, call `dm_ptr_free_ainfo` to free the pointers within `dm_ainfo_struct`

Reading `dm_ainfo_struct` from an `"/ainfo"` group

Call `dm_h5_read_ainfo` to read out an `"/ainfo"` group from an open HDF5 file. It returns a `dm_ainfo_struct`.

At the very end, call `dm_ptr_free_ainfo` to free the pointers within `dm_ainfo_struct`

Jan to write more.

Chapter 3

Procedures for diffraction data analysis

3.1 Cataloging files: the routine `dt_list_series.pro`

Brief description to be written.

3.2 Merging individual ALS 9.0.1 exposures for one 2D view: Henry Chapman's merger

`merger` is an IDL graphical user interface (GUI) program to combine measured diffractograms of different exposure times into one data array. It subtracts CCD background, removes saturated pixels, and masks out the beamstop. There are several ways of starting and getting data into `merger`:

1. Start the program with the IDL command `merger`. Read a `.nc` file (or other compatible format) from the *Intensity data*→*Read frame* menu.
2. Start the program by typing the IDL command `merger`. Read a `.mg` file from the *File*→*Open* menu item.
3. Start the program with the IDL command `merger, info=info`. Insert an array `d` with the command `merger_new, info, alt_data=d`.
4. Start the program with the IDL command `merger, file.mg` where `file.mg` is a `.mg` file that is simply a text file that lists the datafiles and various parameters.
5. Start `merger_manager` with the IDL command `merger_manager, dir=directory`, where `directory` is the directory of datafiles. Then select the files (control-click to select discontinuous lists of files) and click on *Send to merger*.

There are no limits to the number of `merger` windows you can have open.

The main `merger` window lets you browse through the various “frames” of data that will be eventually merged into one array. Select a frame with the slider near the bottom of the window. You can choose whether to display combinations of the intensity file, the background file, background-subtracted intensity, with or without masking of the beamstop mask or saturation. These choices are made from the *Display* panel on the bottom left. Click and drag in the image window to make a rectangular selection. You can then use the *Zoom In*, *Zoom Out*, or *Auto* buttons (the latter displays the image with an intensity scale that spans the minimum and maximum values of that selection). When there is no rectangular selection (a click without a drag), then these buttons act on the entire image. You can do similar things from the *View* menu.

Use the *Background* menu to load in background files. The selection of *Background*→*Auto-background files...* gives a form that will allow newly-read intensity data to automatically choose a background file based on a criterion

of matching a value of a specific `dt_par` tag (most likely `IMAGE_SECONDS`). There is a similar menu and form for Mask files (you could link a mask file to a value of `bsystgm`, for example).

Once you've loaded in all data, background, and masks, you can select *Analyse*→*Merge all frames*. The result of the merge will appear in a new `merger` window.

You can save to a `.mg` file at any time by choosing *File*→*Save As...* or *File*→*Save*. This will produce a text file that you can edit. The file can also be read by some routines. For example,

```
IDL> d = mg_merge('file.mg')
```

will return the merged result of the set saved in `file.mg`.

The `merger` program might seem a bit slow - most of the time is spent on saturation detection, but some of the overhead is on the funny way the program is designed. It is based on another program for doing alignment of optical instruments based on many interferograms, which tried to avoid having too much data in memory and re-reading files when needed.

There is a nice utility called `merger_manager` that gives a database to preview files and list their parameters. From this you can select single or multiple files and send them to `merger` with the *Send to Merger* button. However, I'll first describe a typical session (for a single view) without using `merger_manager`. The session of merging might go something like this:

1. If you haven't already done so, in IDL, type `merger`.
2. Import some data: Choose *Intensity data*→*Read Frame...* and then type in a file or click on the *Browse* button. You may choose more than one file from the list, but for this example just choose one of the files with diffraction data. If it is heavily saturated it will take awhile to load, since it is doing saturation detection.
3. Click on the *Log* button in the lower left to get a better view of the diffraction data.
4. Now set up the background (dark noise) and mask files. For the background, choose *Background*→*Auto-background files*. This allows you to set up a table so that the right background will automatically be subtracted when a new Intensity frame is read in. Since the background is dependent on exposure time, choose `IMAGE_SECONDS` from the *Meta-tag* drop-down list. Click on the *Browse* button to choose a background file. Then click on *Grab selected* which will insert the value of `IMAGE_SECONDS` for the selected background file into the *Value* field. Next, click on *Update table* to enter this into the Auto-background table.

At this point, you can easily set up the other files for other exposures. Click in the table on the second row, since that's where you will want the next entry to go. Edit the filename in the *Filename:* field and then click *Grab selected*. Then click *Update table*. Repeat for the next background exposure: click in the table, edit the filename, click *grab selected* and click *update table*.

Sometimes you might have acquired data files at the same exposure time but with some difference that requires a different background subtraction. For example some of the exposures may be at a different ROI or binning. In this case, using an auto-background based on exposure time only won't be able to distinguish between the two cases. You can add additional conditions for the same background file simply by entering that filename into another row of the auto-background table, and set the tag to `N_ROWS` for example. A background file will only be subtracted from an intensity file only if its metadata satisfies all conditions (ANDed) for that background file.

If you like, click on *Save table* to save the table to a file (useful if this background is the same for another GMR angle). Click on *OK*.

The `merger` program will now chug away for too long (for some dumb reason it recomputes the saturation) and will subtract the background files from all the intensity frames that you have read in (only one, in this example). Note that it is possible to individually select a background for a particular frame (*Background*→*Read background file*); this will always take precedence over the auto-background table.

You will see that the background is subtracted since the listed filename in the main `merger` window (bottom right) has *[B]* after it. Also, the *Background* button in the left *Display* panel is selected. You can click on that button to see the data with background on or off. Or, you can click off the *Data* button to see the background alone.

To scale the greyscale of the data, select a region by clicking in the image window and dragging out a box. Then click the *Auto* button near the bottom.

You can now read in the rest of the intensity data files. Choose *Intensity data*→*Read frame*. . . and then browse to the correct directory etc, highlight the files of interest, and click on *OK*.

Now you want to make some beamstop masks. I do this by choosing the frame that has the longest exposure, going to *Log* mode, turn off saturation, and scaling the data to be able to see the beamstop. Make sure you have a zoom of 1; by default the zoom is 0.5, so click in the middle of the image (but don't drag a box) and then click *Zoom In*. You should see the words *Zoom: 1.000* in the bottom status panel; if not, click *Zoom Out* until you do. Next select *File*→*Save picture*. . . and then choose a filename to save to and a format such as TIFF. I then read this into Photoshop. The picture will be upside-down, but don't worry about it. Then do the following in Photoshop:

- *Image* → *Mode* → *Greyscale*, then *OK*
- *Layer* → *New* → *Layer* then *OK*

Select polygon lasso tool and use it to outline the beamstop. Select a paint brush, choose a really big brush size, black color, and paint inside the selection so it is all black inside the beamstop. In the layers palette, click on the background layer and then choose *Layer* → *Delete* → *Background Layer* *Layer* → *Flatten Image*. You will now have a black and white mask (black inside the beamstop). *File* → *Save As* to save as a tiff file.

Repeat for all beamstop positions.

Now you can set up an auto-load table for the mask as you did for the backgrounds. In *merger*, choose *Mask*→*Auto-mask files*. . . You get a similar window as for backgrounds. This time, you want to tag the files not to *IMAGE.SECONDS* but to *BSYSTGM* or whatever motor distinguishes the beamstop positions. Choose one of the mask files and then click on *Grab current* (the tiff file doesn't know anything about the beam-stop stage, so there's no point choosing *Grab selected*). You need to make sure that the mask you read is the right one for the current frame; the *merger* window is still responsive, so you can use the slider to change to the right frame. Once you've filled in the table, save it to a file and click on *OK*.

Now the *Display* in *merger* should be showing *Data*, *Background*, *Mask*, and *Saturation* all selected. You are now ready to merge the data, which you accomplish by selecting *Analyse* → *Merge all frames*.

Phew!

You will get another *merger* window with the result. This window has two frames: the first (frame 0) is the merged data and the second (frame 1) is the exposure mask (accumulated exposure for each pixel; the units are seconds at 400 mA, that is a value of 1 is equal to 400 mA seconds). You can save these frames individually to datafiles with *File*→*Save data*. . . or save to a picture with *File*→*Save picture*. . . To save all your hard work in assembling the files, go back to the *merger* window with all the intensity frames, and choose *File*→*Save*. Type in a name of a file, for example *gmr00.mg*. This a *.mg* file, or a *merger* file. It is a text file that you can look at (and edit) that has all the appropriate files.

An easy way to merge another similar set of data (e.g., another GMR angle) is simply to copy *gmr00.mg* to *gmr01.mg* (for GMR=1 degree) and change the names in the [Intensity data] section to the new ones. The beamstop masks and backgrounds should still be valid, but if not, edit accordingly. Then, back in *merger*, choose *File*→*Open* to read in that set, and the *Analyse*→*Merge all frames* to get the merged result.

3.2.1 Using merger_manager

Since it gets a bit tedious to select datafiles from a directory listing in a browse window, I wrote *merger_manager*, which gives a nice database and ability to preview files. You can start it with

```
IDL> merger_manager, directory='data'
```

merger_manager can send files to a new *merger* dataset or insert them in any *merger* window (just as if you had read them in from the *Intensity data* menu). In general you will have one *merger_manager* window and possibly many *merger* windows. For dealing with long lists of datafiles, you can filter what is displayed in the *merger_manager* table by setting up a combination of conditions. When acquiring data, you can read the newly acquired files into the table by clicking on *Rescan directories*. You can add directories to be included in the directory scan from the *File*→*Add directory*. . . menu item.

Once you have acquired all data (for example a tilt series) it is helpful to categorise the status of each file. There are a myriad of ways to do this. You can just click on table entries (or use arrow keys) and see in the preview window if the data is background or good data. If it is the first time a file has been selected a thumbnail will be generated, and the total counts is calculated and added to the table. To populate the entire table with the values of total counts you can avoid having to visit every row by clicking on *Generate thumbnails* (and go and enjoy a nice cup of Earl Grey). You can now filter data based on the total counts: e.g., total counts less than a certain number to discriminate background from diffraction, then select all and set the status to *BACKGROUND* using the *Set Status* pulldown list. Or perhaps filter based on ring current.

The *Average selected* button is intended to be used to average background files. This action simply gives an average of intensities, irrespective of exposure time, so it is best to select a group of files of the same exposure time to be used for background subtraction later on. The resulting average is sent to a *merger* window, from where you can save it to a .bin file, using the *File* → *Save data...* menu item.

The *Send to Merger* button will send the selected files to a *merger* window, each as its own frame in the dataset. Remember that you can select any arbitrary group of files by control-clicking on entries in the table. If you have preselected a *merger* template file (a .mgt file) (done by clicking on the *Select .mgt template* button), then this will be applied to the *merger* dataset, but only if you send the files to a new *merger* window. (Actually, it only works if there is no other *merger* window open; I guess that's a bug). The .mgt has all the data of a *merger* dataset except for the individual files themselves; most importantly it may contain the auto-background and auto-mask tables. The template file is created from a *merger* dataset by choosing *File*→*Save template...* If you have defined and selected a template, then the appropriate mask and background subtraction will be carried out when sending files to *merger* from *merger_manager*.

Currently, I have not yet implemented the *Merge and send to Merger* button. It is planned that this will be used to loop over each GMR value, merge all the files of the GMR value (using the .mgt template) and pop the merged result as a frame of a *merger* dataset, with each frame being a different GMR value.

3.2.2 Wish list for enhancements to *merger*

It would be nice to enhance *merger* as follows:

- Put out its results as a "/adi" file.

3.3 Merging individual ALS 9.0.1 exposures for one 2D view: *commie*

The program *COMMIE* is the Cousin of *Merger* for multiple intensity exposures. It is heavily inspired by Henry Chapman's IDL program *merger.pro*, but rewritten from scratch.

The general approach of *COMMIE* is to take a list of individual diffraction exposures (presently only the dt NetCDF diffraction data files taken at ALS beamline 9.0.1 are supported), merge them to avoid pixel saturation and increase dynamic range, subtract out the background, and provide an "assembled diffraction intensities" or /adi group in a HDF5 output file.

This process is driven by a text file (Sec. 3.3.2) that provides the assembly instructions (this file can be generated by the graphical user interface version of *COMMIE*, but the file is still used to guide assembly). *COMMIE* will then make notes on how it produced the assembled diffraction intensity map by writing comments into the /ainfo group within the HDF5 file so that one has a traceable history of the assembly process.

3.3.1 A word on the *commie* distribution

Except for the routines *display_zoom_click.pro* and *ccd_classify.pro* which are in /CVS/diffmic/idl/util, all relevant files are located in /CVS/diffmic/idl/commie.

The routines *display_zoom_click.pro* and *commie_find_bs_widget* rely on a function *getcolor.pro* that comes with the Fanning software package of IDL. I can add it to the *commie* distribution or come up with a different solution if people don't have access to this function.

Updating commie It is desirable that `commie` is updated by as few people as possible assuring that these updates are compatible with the rest of the program. I suggest that, in case of doubt, any requested add-ons are sent to Jan Steinbrener, who will try to implement them in a timely manner. Additionally there is a textfile called `changelog` where any changes should be documented in detail and don't forget to add your initials in case I need to get back to you about it.

The `changelog` file also has a section called 'Latest ideas', where people can add things they would like to see added or changed. I'll try to periodically check this file for new entries. Anyhow, thanks in advance for your cooperation.

Problems with commie If you encounter a bug in while running `commie`, please let me know about all the details that led to that problem, I'll try to fix it as soon as possible. Sometimes there are easy workarounds for bugs, so make sure you check Sec. 3.3.5.

3.3.2 Quick start: Assembly file

This section provides quickstart instructions for using `commie`, many aspects can be fine-tuned by additional keywords in the assembly file, as is explained in more detail in Sec. 3.3.5.

Below is an example of an assembly file that provides `commie` with only the minimum amount of information it needs, the motor positions and where the files are located:

```
high,      bsx=0.01252,bsy=0.015048,dnx=-0.00785,dny=0.009153
top_low,   bsx=0.01254,bsy=0.0136078,dnx=-0.007848,dny=0.009
top_int,   bsx=0.01254,bsy=0.014328,dnx=-0.007852,dny=0.009025
bot_low,   bsx=0.01249,bsy=0.0163141,dnx=-0.007855,dny=0.009282
bot_int,   bsx=0.012555,bsy=0.0153369,dnx=-0.007851,dny=0.009185
bs_moved, bsx=0.01252,bsy=0.0153480,dnx=-0.007850,dny=0.009183
;
topdir,   /common/Users/jsteinbr/data/
sample,   dt_2007_03_09/dt_2007_03_09: 168-287
no_sample, dt_2007_03_09/dt_2007_03_09: 408-527
dark_current, dt_2007_03_06/dark_current/dt_2007_02_28:116-225
```

This is simple enough to write, but a few words are in order on how it is meant to be interpreted:

- One starts with a list of positions (they go on until you hit the character `;`). These can be named anything you want; the end result will be to produce a string array of `position_names=['a_low','a_high','b_low']` These will be used for grouping together files in what follows.
- Following the first position name, a series of motor names are specified, along with the settings of these motors. The above example will produce a string array of `motor_names=['dnx','dny','bsx','bsy']` and for this, the first position name, one will produce floating point array entries of `motor_values[0,0]=30.,motor_values[1,0]=500.,` and so on. That is, `motor_values` will be an array with a number of columns equal to the number of `motor_names` and a number of rows equal to the number of `position_names`.
- When subsequent files are read in, they will be assigned to `position_names` by looking for the greatest number of closest matches to one position. This should normally correspond to the number of specified motors.
- By specifying the `topdir` you provide info on where all data, including masks and `dark_current` files are sitting.
- One then specifies the sequence of files for both the `sample`-in recordings (sample) the `no_sample` recordings of the scattering background, and the dark current exposures (`dark_current`). There is also the possibility

to specify subdirectories to the top directory given in `topdir`. This is illustrated in the sample scriptfile above and keeps the program flexible to accommodate for all sorts of data-sorting preferences.

- There's a subtlety on sequences: it's possible that a data collection run will start late one evening and end the next morning so that the date of the files change. For that reason, it is required that the user always indicates the header of the file right after the subdirectory, as illustrated in the sample scriptfile. The advantage of this over an automated date-rollover routine is that you might have data that is not from subsequent days. **Note** that the header is separated from the indices by the character `:`.
- Specify the dark current files in a similar manner to the sample and `no_sample` files with the tag `dark_current`.

If all information is correct, you can start `commie` with

```
commie, '/path/to/assembly-file'
```

There are some command line arguments that can influence `commie`'s behaviour, see Sec. 3.3.3.

3.3.3 Quick start: Command line arguments (CLAs)

This section is intended to provide an overview over command line arguments accepted by `commie`. For more information on the `/verbose` output see Sec. 3.3.5. At the moment `commie` supports the following command line arguments

- `/verbose`: As the name suggests this will print out details of the assembly process onto the screen or into a file/folder if a filename is supplied, e.g.

```
commie, '/path/to/assembly-file', verbose = '/subdir/verbose.txt'
```

- `/show_final`: This will only display the final assembled array once `commie` is finished.
- `/merge_first`: This CLA will cause `commie` to first merge sample and `no_sample` files separately and then subtract `no_sample` in the very end. The default is to subtract `no_sample` for each position.
- `/no_background`: Choose this option if you do not want `commie` to subtract any `no_sample` files. It will still subtract the dark current though
- `/dk_by_pix`: If this option is selected `commie` will subtract the dark current on a pixel-by-pixel basis. It will also initialize the error arrays by looking at the variance of each individual pixel in the dark current files. Note that this option requires dark current files for each exposure time that the data has been recorded with.
- `/use_saved_merged_pos`: Run `commie` with this option if you wish to read previously saved merged positions from disk. This comes in handy when all you did is update the beamstop mask and wish to have the new mask applied before adding all merged positions together - it will save you a lot of time. Note that the program will check against all CLAs that could potentially modify the outcome of the merging-by-position process (*i.e.* `dk_by_pix`, `merge_first`, and `no_background`) to make sure that they are the same as those that were set when the merged positions were saved to disk.

3.3.4 Quick start: Program flow

This section is intended to provide the reader with a rough idea of how `commie` works. For more in-depth information, see Sec. 3.3.5.

The main idea of `commie` as far as the assembly process is concerned is to make maximum use of all the available data. That means that rather than only patching in the missing beamstop areas from short exposure data, all recorded datafiles are averaged after appropriate normalizations and thresholding. Furthermore, to increase the fidelity of the assembled data, each averaging and normalization procedure is weighted with the absolute errors of each contributing datapoint. That way we can assure that a better known value contributes more towards the final answer than a less well known. Qualitatively, a dataset is assembled in the following steps:

- initialize `commie_script_struct` and `commie_dataset_struct`
- get information from the scriptfile, define defaults
- initialize the dataset, *i.e.* match files to motor positions, determine exposure times
- classify the CCD, *i.e.* determine scaling of dark current centroid and FWHM with exposure time, by either evaluating the dark current files **or** reading the information directly from disk.
- Auto beamstop detection by adding up all images shifted and applying a `region_grow` algorithm.
- merging by position. The most laborious part of the data analysis is described in more detail in Sec. 3.3.5.
- subtract the beamstop
- add up the merged positions
- subtract the merged `no_sample` if applicable
- zero out any negatives
- write out to HDF5 file

3.3.5 Further information

Looking for more information on `commie`? There is a separate manuscript in `/diffmic/doc/commie.tex` that contains all information given above and much, much more detail on how `commie` works and how it can facilitate your daily merging.

3.4 From 2D to 3D data: Anton Barty's program `Xewald`

`Xewald` (pronounced "Ex Ewald") is an IDL program for gridding 2D diffraction pattern data onto Ewald spheres in 3D space. Designed to work with Henry Chapman's `merger` program, reading in processed merger database files and binary merged data files.

3.4.1 Execution

First edit `Xewald` so that the path points to where the `xewald.sav` file is located. Then execute `xewald_startup` to load and run `Xewald`:

```
IDL> .run xewald_startup
```

This will ensure that you get all the necessary subroutines, and the right version of the subroutines in case there are any conflicts in your path. If you want to delve into the source code it can be found in the `src` directory. . .

3.4.2 Using

Most of the options are fairly self-explanatory. . . at least to me.

When first run, the input data will be parsed to compute statistics such as maximum data value.

To avoid doing this each time, add the line `gmax= <value>` to the merger file. This can be done by hand with a text editor, or by saving a new merger file.

3.4.3 Output

For 3D data, output is in 3D format that can be read with `read3d.pro`, e.g.,

```
IDL> a = read3d()
```

The binary file format starts out with the array size n_x, n_y, n_z . The next parameter is the `IDL_data_type`. After that, the file contains a stream of $n_x \times n_y \times n_z$ data pixels of type `IDL_data_type`.

`read3d.pro` contains options to read other types of 3D data file, for example slab decomposition of data output by MPI reconstruction code, and general subregions of a larger cube. The output of `Xewald` is the simplest data format.

For 2D slices through $k_z = 0$, two files are output:

1. The slice is output as a binary GRID file that can be read by `read_grid.pro`:

```
IDL> a = read_grid()
```

This is a legacy file format that I won't explain further here. . .

2. Also output is a 3D slab of data from ± 3 pixels either side of $k_z = 0$, in the 3D format described above. This is to allow you to do your own interpolation or data filling based on what happens either side of $k_z = 0$.

3.4.4 Wish list for enhancements to `Xewald`

It would be nice to enhance `Xewald` as follows:

- Read in 2D `"/adi"` files, and write out 3D `"/adi"` files.
- Besides putting out a 3D `"/adi"` file, write out a `.3da` text file of all the 2D files that went into the 3D file assembly as described in Sec. 2.3. This could be used by `Cewald` as described in Sec. 3.5.1.

3.5 Ideas for programs we should have

3.5.1 `Cewald`: a C program for 3D data assembly

Anton Barty's `Xewald` program provides a beautiful way to interactively assemble 2D diffraction data into a 3D file. However, it would also be useful to have a C program to do this (we might call it `Cewald`, to be pronounced as "See-walled"):

- As one gets to larger and larger data sets, it could be useful to build the 3D array on a parallel computer using MPI-enabled C language programs.
- It's possible that we will encounter a need for refinement of 2D file alignment in the future. One could imagine having a first 3D reconstruction which represents the best current guess of the object based on the totality of all 2D data files. One could then calculate the 3D diffraction intensity corresponding to this reconstruction, and then fit each 2D data file into it to find the best orientation, shift, and scaling for each file. This refinement program could write a `.3da` alignment text file (Sec. 2.3) that `Cewald` could then use to rebuild a 3D `"/adi"` file (Sec. 2.2.2).

3.5.2 Estimating the center of a diffraction pattern

For assembling 2D files into a 3D cube, we want to make sure all 2D files have the center at pixel $n_x/2, n_y/2$. How do we determine this and then shift the files before assembly into the 3D cube? One possible approach is as follows:

1. Obtain a preliminary 2D reconstruction.

2. Do a least squares fit of a plane to the 2D reconstruction to determine the linear phase shift in x and y .
3. Use this linear phase shift to estimate the shift of the center of the diffraction pattern using the shift theorem of Fourier transforms.

3.5.3 Estimating information content in a 2D view

For a given resolution annulus, compare variance of signal with expected shot noise variance? Measure mean speckle size and contrast?

3.5.4 Characterizing noise

To what extent is this done by `merger`?

3.5.5 Masking the beamstop and detecting saturated pixels

To what extent is this done by `merger`? Have `merger` able to calculate the data error factor array of Sec. 2.2.3?

3.5.6 Displaying autocorrelation function from subregions of the diffraction data array

As demonstrated by Stefano Marchesini.

3.5.7 Estimating support from the autocorrelation

Write a support file (Sec. 2.2.6).

Chapter 4

Iterative phasing programs

4.1 Pierre Thibault's `retriever`

`retriever` is a very simple program implementing Veit Elser's difference map algorithm for 2D densities. It was written by Pierre Thibault at Cornell. It is located in the CVS archive at `diffmic/c/retriever`. What it does:

- Compute a predetermined number of iterates of the difference map, starting with either a random or a previously saved iterate.
- Any pair of projections can be used, and adding new projections is a hopefully straightforward process.
- Parameters are stored in XML format. The global function `get_parameter` provides access to all parameters (required by both `retriever` and any user-provided projections).

What it does not do:

- Averaging, modes, or any other such fancy stuff.
- It does not provide the possibility of dumping an array during the run; that should be added very easily.
- The Input/Output formats do not comply with the `"/adi"` format. This would require just a bit of fiddling in the file `io.c`.

Compatibility on different systems has not been checked. I just know it compiles on my laptop (running Linux Debian unstable). Among the possible issues, the Makefile might not work with non-GNU make. The libraries required are `libxml` and `wfft3`, both open-source and easy to install.

4.2 Anton Barty's `r3d.mpi`

Anton Barty has written a very impressive program for 3D reconstruction using MPI-enabled C code on Apple clusters. His program `r3d.mpi` is documented in the CVS file `diffmic/c/r3d.mpi/README.pdf`.

4.2.1 Wishlist for `r3d.mpi`

Here is a wishlist for enhancements to `r3d.mpi`:

1. Allow file I/O to work with `"/adi"`, `"/spt"`, and `"/itn"` files as described in Chapter 2.

4.3 Wishlist for reconstruction programs

4.3.1 Application of Fourier modulus constraint with error factor per pixel

The error factor is described in Sec. 2.2.3.

4.3.2 Application of support constraint

4.3.3 Calculation of the next iterate

Calculation of the next iterate given the previous one and the parameters β , γ_1 , and γ_2 .

4.3.4 Calculation $I_{\text{recon}}(f)/I_{\text{data}}(f)$ at a particular iterate

Depends on spatial frequency. Save it to a `.csv` (comma, space, value) ASCII text file so both Excel and IDL can read it.

4.3.5 Calculation of error metric at a particular iterate

Do we just use an average over all spatial frequencies f of $I_{\text{recon}}(f)/I_{\text{data}}(f)$? Or do we use some other more traditional error metric?

4.3.6 Script language for running iterations

We want to be able to write some sort of text file that tells the algorithm what `/adi` and `/spt` files to read in, what parameters to use for how many iterations, how often to spit out a `/itn` file, and so on. Should this be a Python file?

4.3.7 Structure outside the support

Can we estimate how much scattering might still be in the regions we have assigned to be outside the support constraint? Pierre Thibault and Veit Elser have done something along these lines.

Chapter 5

Visualization of results

5.1 Routines for making movies of iterates

Anton Barty has done something about this. . .

5.2 Routines for viewing 3D data

Include viewing cutaway surfaces, movies of translucent structure, etc.

Chapter 6

The array calculation subroutine library `dm_array.c`

Besides the `dm_fileio` routines described in Sec. 2.4 which are used for reading and writing files, the core routines are those in `dm_array.c` in CVS `diffmic/c/util`. These routines handle operations on complex array structures (see Sec. 1.4.6), and they are meant to all be MPI-enabled for parallel computation if they are compiled with `cc -D__MPI__`.

The best source of documentation on these routines is the information contained in the header file `dm_array.h`, where (so far at least) every routine is preceded by a short description of what it does. So far they all follow a certain programming pattern:

- In general they all work on 3D complex array structures, following the mechanisms for creation and access (through `c_re` and `c_im`) described in Sec. 1.4.6. If your array is really just a 2D array, that's OK; just set `nz=1` (and for a 1D array set `ny=1` and `nz=1`).
- For consistency with the above convention for complex arrays, complex scalars are passed to and from the routines as complex arrays (Sec. 1.4.5) with one element.

As a short example, here's how you would use these routines to create an array, fill it with a Gaussian, and add a real number of 0.5 into the array:

```
dm_array_complex_struct array_3d_cas;
dm_array_index_t ipix;
dm_array_real this_re, this_im;

array_3d_cas.nx = 512; array_3d_cas.ny = 512; array_3d_cas.nz = 512;
array_3d_cas.npix = array_3d_cas.nx*array_3d_cas.ny*array_3d_cas.nz;
DM_ARRAY_COMPLEX_MALLOC(array_3d_cas.complex_array, array_3d_cas.npix);
array_3d_cas.ptr_forward_plan = (dm_fft_plan *)
    malloc(sizeof(dm_fft_plan));
array_3d_cas.ptr_inverse_plan = (dm_fft_plan *)
    malloc(sizeof(dm_fft_plan));

sigma_x = 0.1*nx;  sigma_y = 0.2*ny;  sigma_z = 0.3*nz;
dm_array_load_gaussian(&array_3d_cas, sigma_x, sigma_y, sigma_z);
dm_array_add_real_scalar(&array_3d_cas, (dm_array_real)0.5);
```

And of course at the end of whatever routine this was in, you'd want to do `DM_ARRAY_COMPLEX_FREE(complex_array)`; to avoid memory leaks...

6.1 MPI-enabled FFTs

Because the data is distributed among many processors on an MPI-enabled machine (see *e.g.*, http://www.fftw.org/fftw2_doc/fftw_4.html#SEC58), one must work with the data in a different fashion. Each MPI process will call the same FFTW routines (each of which is a MPI “blocking” routine), but each process will operate on a different part of the data.

6.1.1 Working with FFTW 2.1.5 MPI

This is how you do it in FFTW 2.1.5 MPI for an array with real-space dimensions $[n_x, n_y, n_z]$, where n_x is the “fast” index for a “row-major” array as described in Sec. 1.4.4 (we’ll also assume `n_fields=1`).

1. Following the usual call to `MPI_Init()`, you call

```
fplan = fftw3d_mpi_create_plan(mpi_comm, nx, ny, nz,
    FFTW_FORWARD, flags);
if (output_order == FFTW_NORMAL_ORDER) {
    iplan = fftw3d_mpi_create_plan(mpi_comm, nx, ny, nz,
    FFTW_INVERSE, flags);
} else {
    iplan = fftw3d_mpi_create_plan(mpi_comm, ny, nz, nx,
    FFTW_INVERSE, flags);
}
```

2. You next call the routine

```
fftwnd_mpi_local_sizes(fplan, &local_nz, &local_z_start,
    &local_ny_after_transpose, &local_y_start_after_transpose,
    &total_local_size);
```

to get information on the part of the data that you will be working with. You must then use `malloc()` to create `total_local_size` pixels of complex data (we’ll call this sub-array `local_data`), and an equal amount of workspace array data (we’ll call this sub-array `workspace`).

3. You can then load your part of the data. Let’s say you have a function `f(i)` which provides the initial value for each pixel in a 3D real-space array with `i=ix+iy*nx+iz*nx*ny`. You would load data into this array in real space with

```
for (iz=0; iz<local_nz; iz++)
    for (iy=0; iy<ny; iy++)
        for (ix=0; ix<nx; ix++)
            i = ix+iy*nx+(iz+local_z_start)*nx*ny;
            c_re(local_data, i) = f(i);
```

4. You then do a transform on the data with

```
fftwnd_mpi(plan, n_fields, local_data, workspace, output_order);
```

You can then refer to Fourier space data in the output as

```
if (output_order == FFTW_NORMAL_ORDER) {
    for (iz=0; iz<local_nz; iz++)
        for (iy=0; iy<ny; iy++)
```

```

        for (ix=0; ix<nx; ix++)
            i = ix+iy*nx+(iz+local_z_start)*nx*ny;
            this_real = c_re(local_data,i);
    } else {
        for (iy=0; iy<local_ny_after_transpose; iy++)
            for (iz=0; iz<nz; iz++)
                for (ix=0; ix<nx; ix++)
                    i = ix+iz*nx+(iy+local_y_start_after_transpose)*nx*nz;
                    this_real = c_re(local_data,i);
    }
}

```

5. At the end of the program, you should call `fftwnd_mpi_destroy_plan()` on `fplan` and `iplan`, and `MPI_Finalize()`.

6.1.2 Working with `dist_fft`

The normalization of data array conserves for both forward and backward `dist_fft`. And array is not centered after `fft`.

This is how you do it in `dist_fft`. Note that the `dist_fft` routines require you to have $n_x = n_y$ for 2D, and $n_x = n_y = n_z$ for 3D.

1. Following the usual call to `MPI_Init()`, call

```

planForward = dist_fft_create_plan(type, dimension, DIST_FFT_FORWARD,
                                forwardFlags, comm);
planInverse = dist_fft_create_plan(type, dimension, DIST_FFT_INVERSE,
                                inverseFlags, comm);

```

2. Then allocate local space by calling

```

dist_fft_local_dimensions(planForward, &local_dimension, &local_start,
                          &local_storage_size);
DIST_FFT_MALLOC_DATA(data, local_storage_size);

```

The initial array is truncated along the last dimension. For example, if a $9 \times 9 \times 9$ array is split into 3 processes, the array size for each process is $9 \times 9 \times 3$, `local_dimension` should be 3, and `local_start` should be 0, 3, 6 respectively. For convenience we also allocate a spare space with the same size, which will be used by transpose operation etc.

```

DIST_FFT_MALLOC_WORKSPACE(workspace, local_storage_size);

```

3. We can load parts of our data now. Let's say we have a function `f` which provides the initial value for each pixel in a 3D real-space array with. We would load data into this array in real space with

```

start = local_start * nx * ny
limit = (local_start + local_dimension) * nx * ny
FOR(index = start; index < limit; index++){
    local_index = index - start;
    c_re(data, local_index) = c_re(f,index);
    c_im(data, local_index) = c_im(f,index);
}

```

4. We then do a transform on data with

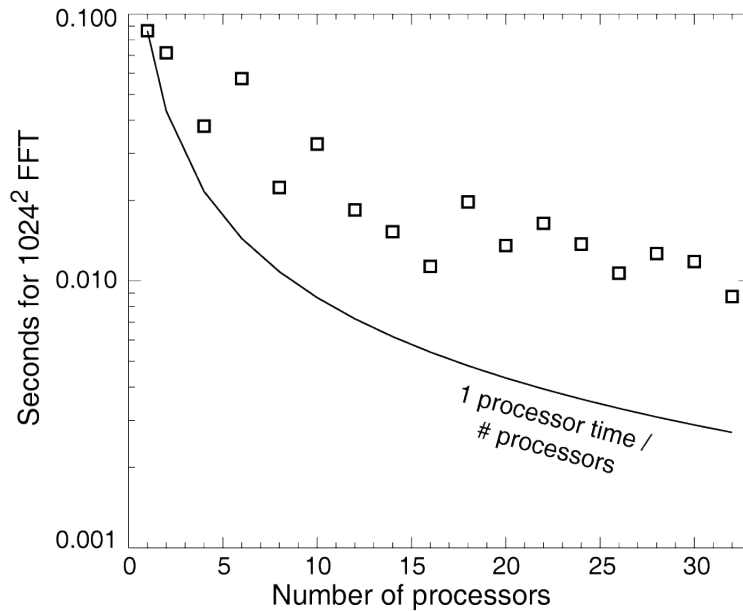


Figure 6.1: Benchmark times for running `dist_fft` on a 2D FFT using varying numbers of processors.

```
dist_fft_execute(planForward, data, workspace);
dist_fft_execute(planInverse, data, workspace);
```

5. At the end of the program, call

```
DIST_FFT_FREE_WORKSPACE(workspace);
dist_fft_destroy_plan(planForward);
dist_fft_destroy_plan(planInverse);
MPI_Finalize();
```

`DIST_FFT_COLUMN_INPUT` refers to array indexing with x being the fast array index. Test results on the Stony Brook cluster for `dist_fft` FFTs, using 32 processors:

N^3	data	order	input	output	Seconds
1024^3	float	split	COL	COL	13.4
1024^3	float	interleaved	COL	COL	18.0
512^3	float	split	COL	COL	4.1
512^3	float	interleaved	COL	COL	3.5
512^3	float	split	ROW	ROW	8.5
512^3	float	interleaved	ROW	ROW	6.8
512^3	float	split	ROW	COL	6.4
512^3	float	interleaved	ROW	COL	5.2
512^3	float	split	COL	ROW	6.2
512^3	float	interleaved	COL	ROW	5.2
512^3	double	split	COL	COL	6.6
512^3	double	interleaved	COL	COL	4.4

6.2 FFTs using `dm_array_fft()`

The library `dm_array.c` provides unified access to several different FFT routines through the routine `dm_array_fft()`. This routine can be trusted to preserve energy in both forward and inverse transforms (meaning that the value of the sum of squares of all amplitudes is not changed). This routine is just a “wrapper” for one of several underlying FFT routines. FFTW3 (www.fftw.org) is reputed to be the fastest of the single-processor FFT routines available, so it is one good choice. However, FFTW3 is not MPI-enabled as of yet, although an older version (FFTW 2.1.5) provides this capability and is still available. Apple has apparently used FFTW 2.1.5 as the basis for a very fast library for Xserve G5 clusters called `dist_fft`; this library is MPI-enabled and it also exploits the pipelining features of the altivec unit on G5 processors. The performance of Apple’s `dist_fft` routines is discussed in a white paper which we have copied into `GigaFFTonG5.pdf` in CVS `diffmic/doc/recon`, and they are available from the `dist_fft` web page. There are two or three steps to using `dm_array_fft`:

1. You must first allocate memory for the complex array structure using the mechanism described in Sec. 1.4.5. (And remember that these routines treat 2D arrays as 3D arrays with $nz = 1$). You must then create a “plan” (a set of precalculated parameters and radix choices) for forward and inverse transforms:

```
MPI_Comm mpi_comm;
dm_array_fft (&array_3d_cas,
              (DM_ARRAY_CREATE_FFT_PLAN | DM_ARRAY_FFT_MEASURE),
              mpi_comm);
```

The act of creating a plan destroys any data in `complex_array`, so you should create your plan *before* you start to load data into `complex_array`.

When creating a plan, you can trade off slow, exhaustive testing of the best optimization strategy so as to do the fastest FFTs (`DM_ARRAY_FFT_PATIENT`), a balanced mix of faster planning and somewhat slower FFT execution (`DM_ARRAY_FFT_MEASURE`), and fast planning with slower FFT execution (`DM_ARRAY_FFT_ESTIMATE`); the default type is `DM_ARRAY_FFT_PATIENT`.

2. You can then happily continue to do in-place forward and inverse FFTs on the same array:

```
dm_array_fft (&array_3d_cas, DM_ARRAY_FORWARD_FFT, mpi_comm);
dm_array_fft (&array_3d_cas, DM_ARRAY_INVERSE_FFT, mpi_comm);
```

3. If you change the array dimensions $n_x \times n_y \times n_z$, you must destroy the plan, free and then reallocate the array memory using the mechanisms described in Sec. 1.4.5, and create a new plan for the new array size. The way to destroy an existing plan is

```
dm_array_fft (&array_3d_cas, DM_ARRAY_DESTROY_FFT_PLAN, mpi_comm);
```

FFTW (non MPI) is used by `dm_array_fft()` as default, unless `-DDIST_FFT` is specified while compiling to use Apple `DIST_FFT` routines. Doing FFT with FFTW is tested in Cygwin environment. The routine takes care of renormalization and recentering array. Working with `DIST_FFT` is tested on Apple Cluster up to 32 processes (for details, please refer to Appendix B). The routine renormalizes FFT result, but does not recenter array, which is shifted by $n/2$ along X, Y and Z directions.

Chapter 7

A roadmap for work at Stony Brook

Here are some thoughts for a way to proceed at Stony Brook. We should keep in mind the C coding conventions described in Sec. 1.4. The subroutine libraries can draw inspiration from the code of Anton Barty's `r3d_mpi` program (Sec. 4.2), and Pierre Thibault's `retriever` (Sec. 4.1) program. Characteristics we'd like to have for our libraries include:

- The ability to compile on both Apple OS X and Linux systems, with and without MPI capabilities, with changes only in a `Makefile` or by adding something like an environment variable or a `-D__MPI__` type argument with the `cc` command.
- Generation of better `Makefiles` using rules to simplify them.
- Work with the `"/adi"`, `"/spt "` `"/itn"`, and `.3da` files described in Sec. 2.2.
- Have testing programs provided to verify/demonstrate the correctness of major operations. These should go into a subdirectory `test` below the directory of the routines they are to test.

Things to be done:

1. I think the style in which the first few MPI-enabled routines are written is very important for establishing the standards for the package. If the first routines are well written, then we will be in good shape for building the package further. Therefore we should get Nick D'Imperio to help with tweaking the first, simple routines in `dm_array.c` before going too much farther.
2. Finish programming and testing the routines for reading and writing assembled diffraction intensity or `"/adi"` files (Sec. 2.2.2). Add to the documentation of `"/adi"` files the text strings that are added in to track operations on a file. Jan Steinbrener.
3. Once this is done, go on and do the same for support mask `"/adi"`, `"/spt "` (Sec. 2.2.6) files, and iterate amplitude `"/itn"` files (Sec. 2.2.7). Perhaps Huijie and Xiaojing should each tackle one of these formats to make sure they understand the details of file I/O?
4. The routine `dm_array_test.c` in CVS `diffmic/c/util/test` is meant to exercise `dm_array_fft()` and verify its output. I don't think the PNG images of slices taken out of the diffraction cube look right. This needs to be looked into.
5. We need to add the code into `dm_array_fft()` to properly invoke the Apple `dist_fft` routines, and test it.
6. We should have a routine that is the equivalent to `shift(array, nx/2, ny/2, nz/2)` in IDL. It might already exist in `dist_fft`?
7. We want routines like `dm_pi_modulus`, `dm_pi_support`, and `dm_dmap_delta` to calculate the modulus and support constraints, and calculate the difference Δ in the difference map algorithm. The routine `dm_dmap_delta` should return the error measure ϵ .

Appendix A

Running MPI code

A.1 Running code using LAM MPI on Stony Brook's Apple cluster

It appears that LAM MPI is the best choice for Stony Brook's Apple Xserve G5 cluster. Here is some information on how to build and run LAM MPI programs.

A.1.1 Things to set up just once

The first step is to make sure that you have `ssh` set to work to get to the other nodes without requiring a password by using a public/private key pair. To do this, do `ssh-keygen -t dsa`. Accept the default key file of `id_dsa` in your `~/.ssh` directory (overwriting a previous version if necessary), and hit *enter* twice to have a blank passphrase. You'll then get a message such as

```
Your identification has been saved in /home/mcuttler/.ssh/id_dsa.
Your public key has been saved in /home/mcuttler/.ssh/id_dsa.pub.
The key fingerprint is:
4a:05:dd:83:72:c1:0e:f2:05:a0:80:cd:81:ac:e3:4f mcuttler@cuttler
```

You should then do the following two commands:

```
[mcuttler@cuttler ~]$ cat .ssh/id_dsa.pub >> .ssh/authorized_keys
[mcuttler@cuttler ~]$ chmod -R go-rwx .ssh
```

Now, `ssh` from the master node to each of the other nodes using the fully qualified domain name (*i.e.*, `node001.cluster.private` through `node015.cluster.private`). This will set some necessary entries into `~/.ssh/known_hosts`; as a result, `ssh` should now work without you needing to enter any type of password.

The next step is to make your life slightly easier by making a simple script to point to the right version of the program `mpirun`. What you should do is to modify your file `.bash_profile` to have both your present working directory (`.`) and your own `bin` directory in the `PATH`:

```
export PATH=.:~/bin:$PATH
```

(you can make this active the first time by typing `. ~/.bash_profile` but in subsequent logins you can skip this step). Now create the `bin` directory using `mkdir`, and create a file there called `lamrun` which looks like this:

```
#!/bin/bash
/usr/local/bin/mpirun $@
```

Once you've created this file, do `chmod u+x lamrun` to make it executable.

You should now make a file called `machines` in whatever directory you wish to run a program in. This is a simple text file which should read

```
portal2net.cluster.private
node001.cluster.private
  (and 002-014)
node015.cluster.private
```

so that the file has 17 lines.

You should also add something like the following to your Makefile:

```
OS      = $(shell uname -s)
KARCH   = $(shell uname -m)
ifeq ($(OS), Darwin)
  CC     = /usr/local/bin/mpiCC
  CFLAGS = -O3 -mcpu=G5 -mtune=G5 -mpowerpc64 \
           -faltivec -fstrict-aliasing
endif
```

Your *Makefile* will then have lines like

```
hello.o:      hello.c
              $(CC) -c hello hello.c $(CFLAGS)
```

so that it will use the proper type of C compiler and flags (and remember that Makefiles require TAB characters for indenting).

A.1.2 Running your code

Now that you've done the above setup steps, you are ready to run MPI programs. To do so, you must first start the LAM servers:

```
lamboot -v machines
```

You can then run your program with

```
lamrun -np 32 myprogram
```

where of course you can specify fewer than 32 processors. After your job is finished you must shut down the lam servers:

```
wipe -v machines
```

A benchmark of how the `dist_fft` routine works with varying numbers of processors is shown in Fig. 6.1.

A.2 Running `dist_fft_test` using MPICH

It appears that LAM MPI works better on the Altivec cluster; follow the instructions in Sec. A.1 instead. Here are the steps to compile and make `dist_fft_test` on `altivec` from the CVS directory `c/dist_fft`:

- Make sure your `.bash_profile` file in your home directory has the following lines:

```
export PATH=/usr/local/mpich-1.2.7/ch-p4/bin:$PATH
export P4_RSHCOMMAND=rsh
```

These might be in addition to your other lines needed for CVS of

```
export CVS_RSH=ssh
export CVSROOT=':ext:diffmic@xray1.physics.sunysb.edu:/home/xray1/diffmic/'
```

The first time you enter this in you'll have to type `. .bash_profile` to make it active; it will be there automatically for subsequent logins.

- Do `cp Makefile.altivec Makefile` to use the correct makefile. You also need to have the file `mach` in the directory but this should be there automatically from CVS.
- Type `export CC=mpicc, and make dist_fft_test`.
- Now you're ready to run! When running the program, you need to provide the \log_2 dimensionality of the calculation. The calculation will be run with this same number of points in 1D, 2D, and 3D, so if you specify 12 you will get a 1D FFT of size $2^{12} = 4096$, a 2D FFT of size $2^6 \times 2^6 = 64 \times 64$, and a 3D FFT of size $2^4 \times 2^4 \times 2^4 = 16 \times 16 \times 16$. You can also specify the number of processors with the `-np` flag: `mpirun -np 2 -machinefile mach dist_fft_test 12`
- To change to Non-Interleaving mode or Double precision you must modify the file `dist_fft_types.h` by modifying the following lines:


```
// the following options are off by default
// #define DIST_FFT_USE_INTERLEAVED_COMPLEX
// #define DIST_FFT_USE_DOUBLE
```

A.2.1 Running MPICH code on a subset of the cluster

The cluster has been split in to 4 subsets using 4 `mach` files named `mach1`, `mach2`, `mach3` and `mach4`. Each `mach` file has a list of processors that can be used as a given subset. Here is an example of how to run the `dist_fft_test` on a subset of the apple cluster:

- Log onto master node of cluster
- Set Paths and compile program (see previous section for details)
- From the master node, `ssh` to a node in the subset you wish to work with


```
ssh node005.cluster.private
```
- The home directories are available on all the nodes. Move to the directory containing your code:


```
cd /common/Users/astewart/diffmic/c/dist_fft
```
- Run the program as before, using the `mach` file appropriate to your subset of processors to be used:


```
mpirun -np 2 -machinefile mach2 dist_fft_test 12
```

Appendix B

Running MPI DIST_FFT

B.1 Compiler

The MPI version is LAM MPI on Cluster now. Make sure to use LAM MPI compiler. It appears that `/usr/local/bin/mpiCC` links to MPICH compiler. Using `$locate mpicc`, we can find one LAM compiler at:

```
/Volumes/RAID/common/mpich/lam-7.1.2/bin/mpicc.
```

B.2 Header file modification

Compiling `dm_array_fft()` with LAM MPI `mpicc`, it stops with an error message complaining that there is an invalid Boolean type definition in the following system header file:

```
/System/Library/Framework/vecLib.framework/Headers/vecLibTypes.h
```

Considering we don't have authority to modify this header file and we don't really use Boolean type variables, we can cope the header file into local directory, change links combined with the file, and command out that error-causing line.

The error message is removed. And `dm_array_fft()` works up to 32 processes.