ADVANCED SEMINAR SERIES
FRESNEL COHERENT DIFFRACTIVE IMAGING
COMPUTATIONAL LAB

Nadia Davidson

The University of Melbourne

Tuesday, 22nd March 2011
Overview

What I plan to cover:

- Is should be achieve in the lab?
- Introduction to the software you’ll be using
- A bit about C/C++
- How to use the code to perform a Fresnel CDI simulation

These notes are available on the website in case you need to look things up during the lab. Go to “Presentations” and select 20110321_cxs_lab.pdf

Please stop me if you don’t understand something.
Your Project

Simulation of Fresnel Coherent Diffractive Imaging Experiment

Parameters: Zone plate: diameter = 160 microns, outer zone = 50 nm. energy = 2500 eV. Sample to Detector = 1m.

Create a transmission function of an arbitrary object. Propagate Exit Surface Wave to Detector. Recover Object.

Need to calculate: Sampling ratio, Fresnel number across the object, Resolution, actual sample dimensions (need to know pixel size in sample plane).

Reading:
Fresnel coherent diffractive imaging: treatment and analysis of data, New Journal of Physics, 12, 035020
Coherent diffractive imaging using short wavelength light sources, Journal of Modern Optics, 57, 13, 1109-1149
Contents of the Report

What you need to do:

1. Construct complex transmission function for single element 'thin object' - should be different from the example.
2. Make a forward simulation, obtain the intensity and reconstruct the object using a 'loose support', a support the exact size of the object and a support which is too small.
3. In each case plot the error metric showing the progress of the reconstruction at different stages.
4. Present a comparison of the recovered object and the original with line outs.
5. Estimate the resolution from the reconstructed image and compare to the theoretical value (e.g. Abbe criterion).
6. Include these images in a short report as if this were a real experiment. What were the key parameters: Fresnel number, sample to focus, how big would your object really be?

The report should be approx. 3 pages in length.

It is due on the 1st of April (the Friday after next)
**Introduction to the CXS software**

**What is the software?**
- A C++ library which users can import into their own c code
- A set of IDL routines
- Command line tools

**What tasks can be performed?**
- Plane wave coherent diffractive imaging (Planar CDI)
- Fresnel coherent diffractive imaging (FCDI)
  - Reconstruction of the white-field
  - Reconstruction of the exit-surface wave and transmission function of the sample

**What examples are provided?**
- Planar reconstruction (with real data)
- Planar simulation (and simulated data reconstructed)
- Fresnel white-field reconstruction (with real data)
- Fresnel reconstruction (with real data)
- Fresnel simulation (and simulated data reconstructed)

**What options are there for the reconstruction?**
For the reconstruction, you have various options such as:
- changing between different algorithms (e.g. HIO to ER)
- updating the support or using the shrink-wrap algorithm
- starting with any exit-surface wave initialisation you like (e.g. reload a result from a previous reconstruction), or you can use the default initialisation.

Red means that these bits are relevant to your lab.
write_tiff("trans_phase_input.tiff", result);

/************************ now simulate: ******************************************/

// get the illuminating white field in the sample plane
proj.propagate_from_detector(wf);

// multiply the transmission function by the white field
input.multiply(wf);

// get the magnitude of the wave
input.get_2d(MAG_SQ, result);

// write the output to file
write_tiff("sample_plane.tiff", result);

// propagate to the detector
proj.propagate_to_detector(input);

// take the wf back to the detector so we're ready for the reconstructions
proj.propagate_to_detector(wf);

// write input intensity to Double_2D array
// "result" will be used as the diffraction data.
input.get_2d(MAG_SQ, result);

// write the diffraction pattern to a file
write_tiff("forward_projection.tiff", result);

// if you feel like making a more realistic simulation with thresholding

A bit about C/C++

The basic structure of a C program will look like this:

```c
/**
 * This is a comment describing the code
 */

#include <iostream>
#include <sstream>
#include <stdlib.h>
#include "io.h"

int main(int argc, char * argv[]){
    //declare some variables
    Double_2D object;
    std::string file_name = "FCDI_simulation_object.tiff";
    std::string file_name_log = "FCDI_simulation_object_log.tiff";

    //load an image file for the sample
    read_image(file_name, object);

    //write on log scale
    write_image(file_name_log, object, true);

    return 0;
}
```

- **include** - makes the functions declared in the header file available in this code.
- **int main(int argc, char * argv[])** - defines the main function. This is a special function that will be executed when you run the code.

The int at the beginning defines the type which main will return, the (int argc, char * argv[]) defines the types which the function takes. You shouldn’t need to worry about this for the lab.

- **Double_2D object; and std::string file_name = "FCDI_simulation_object.tiff" declare some variables.**

The first part (Double_2D and std::string) are types.

The second part (object and file_name) are the names of the variables. i.e. what you call them in the code.

For many variables you can set their value when you declare them as well. Note: you can declare a variable anywhere in the code, it doesn't have to be at the beginning of the function.

- **read_image(file_name, object);** - perform an operation. In this example we read an image file and put it’s contents into an object of type Double_2D.
The CXS software library contains lots of functions you can use:

- Some are C code such as:
  ```c
  read_image(file_name, object);
  ```

- Many are C++ such as:
  ```cpp
  Double_2D image_1;
  Double_2D image_2;
  ..... set the Double_2Ds by reading images
  image_1.add(image_2);
  ```
  (In C++ the function belongs to the variable, so you have to use the syntax: `<variable_name>_<function>`) 

You will need to use the classes (C++ object types):
- Double_2D
- Complex_2D and
- FresnelCDI

The example demonstrates how to use them.
FresnelCDI Class Reference

The class which performs Fresnel CDI reconstruction. More...

#include <FresnelCDI.h>

Public Member Functions

FresnelCDI (Complex 2D &initial_guess, Complex 2D &white_field, double beam_wavelength, double focal_detector_length, double focal_sample_length, double pixel_size, double normalisation=1.0, int n_best=3)

virtual ~FresnelCDI ()
virtual void initialise_estimate (int seed=0)
virtual void get_transmission_function (Complex 2D &result, bool inforce_unit_mag=true)
virtual void scale_intensity (Complex 2D &c)
virtual void propagate_from_detector (Complex 2D &c)
virtual void propagate_to_detector (Complex 2D &c)
void set_normalisation (double normalisation)
void set_experimental_parameters (double beam_wavelength, double focal_detector_length, double focal_sample_length, double pixel_size)
virtual int iterate ()

Complex 2D * get_best_result (double &error, int index=0)

void set_support (const Double 2D &support, bool soften=false)
void set_intensity (const Double 2D &detector_intensity)
void set_beam_stop (const Double 2D &beam_stop_region)
void set_relaxation_parameter (double relaxation_parameter)
void get_intensity_auto_correlation (Double 2D &auto)
void set_algorithm (int alg)
void set_custom_algorithm (double m1, double m2, double m3, double m4, double m5, double m6, double m7, double m8, double m9, double m10)
void print_algorithm ()

double get_error ()

virtual void apply_shrinkwrap (double gauss_width=1.5, double threshold=0.1)

void get_support (Double 2D &support)
virtual void apply_support (Complex 2D &c)
virtual void project_intensity (Complex 2D &c)
void set_fftw_type (int type)
void set_complex_contrast_function (void*complex_contrast(Complex 2D &transmission))

Static Public Member Functions

static int getAlgFromName (std::string algorithm_name)

Protected Member Functions

void apply_threshold (Double 2D &array, double threshold)
Doxygen cont.

Static Protected Attributes

\[
\text{static std::map< std::string, int > \text{algNameMap} = PlanarCDI::set_up_algorithm_name_map();}
\]

Detailed Description

Author:
Nadia Davidson <ndavidson@unimelb.edu.au>

The class used for performing Fresnel CDI reconstruction (for white-field reconstruction see FresnelCDI WF), it inherits most methods from PlanarCDI, so please look at the documentation of this class also. Although there are some differences in the underlying code between this class and the planar case, the interface is generally unchanged. Therefore users should refer to the instructions for PlanarCDI to understand how to use a FresnelCDI object in their own code. Only the differences relevant to users will be documented here.

Definition at line 27 of file FresnelCDI.h.

Constructor & Destructor Documentation

FresnelCDI ( Complex 2D & initial_guess, Complex 2D & white_field, double beam_wavelength, double focal_detector_length, double focal_sample_length, double pixel_size, double normalisation = 1.0, int n_best = 1 )

Create a FresnelCDI object. All of the dimensions which are given as input should be in the same units, e.g. metres.

Parameters:
- initial_guess: The complex 2-D field which is modified by this object. This represents the exit-surface-wave of the sample. It may be pre-initialised to a best first guess (e.g. manually); it may be loaded from a file (e.g. the output from a previous reconstruction job); or it may be uninitialised, and the default initialisation provided by FresnelCDI may be used.
- white_field: The reconstructed white-field (which has been previously determined using FresnelCDI WF or otherwise)
- beam_wavelength: Wavelength of the beam
- focal_detector_length: The distance from the focal point to the detector
- focal_sample_length: The distance from the focal point to the sample
- pixel_size: The diameter of one of the detector pixels.
- normalisation: The amount to scale the data before subtracting the white field illumination. By default this is 1.0.

Definition at line 16 of file FresnelCDI.cpp.

References Complex 2D::copy(), illumination, pixel_length, PlanarCDI::set_algorithm(), set_experimental_parameters(), and wavelength.

virtual FresnelCDI() [inline, virtual]

The destructor for this class.
An aside: Compiling and running (not necessary for the lab)

- C and C++ programs must be compiled before you can run them.
- In general, you can compile a C/C++ program with the command `g++` (or `gcc` if it’s only C code):
  ```
g++ -o hello.exe hello.c
  ```
- However, if you’re linking with external libraries (for example, the library for the cxs software) is a bit more complicated:
  ```
g++ -o hello.exe -L<directory_to_the_library> -l<library_name> -I<directory_to_the_library_header_files> hello.c
  ```
- You can see that this gets long and complicated if you have to link with many libraries. Therefore, people often put these commands into a Makefile.
- The major advantage of using make is that it keeps track of when changes are made to your code and only recompiles if a change was made to the source code.
In the lab, you can use the Makefile provided (it’s in the same directory as the example code)

- if you type `make`, all the examples which come with the software will compile
- if you type `make clobber` the executable files will be deleted. Hence typing `make clobber` followed by `make` forces a recompilation.
- if you want to add a file to be automatically compiled with `make`, open the Makefile and edit the variable `EXAMPLE_SRC` by adding your `.c` filename to the end of the list.
Instructions to get started

1. log in to one of the baker linux machines
2. open a terminal and make a directory for working in
3. change into the new directory
4. copy the examples directory from my home area to your working area:
   ```
   cp -r /home/ndavidson/cxs/compiled_rel_0.2.5/cxs_software/examples ./
   ```
   This directory contains:
   - example code (including FresnelCDI_simulation_example.c)
   - a Makefile
   - a white-field file (wf_reconstructed.cplx), created by reconstructing some white-field data.
5. enter the examples directory and compile everything by typing `make`.
6. run the Fresnel simulation example to see what it does:
   ```
   ./FresnelCDI_simulation_example.exe
   ```
7. read and edit the Fresnel simulation example code using a text editor of your choice e.g:
   ```
   emacs ./FresnelCDI_simulation_example.c
   ```
8. recompile the code by typing `make` again.

Some basic linux command you might need are listed in a back-up slide.
Good Luck!
Commonly used Linux/Unix commands

- `mv file1 file2` - move file1 to file2 (this can also be used for renaming files).
- `cp file1 file2` - copy file1 to file2 (file1 will be deleted).
- `mkdir directory` - make a directory
- `ls` - list the files in the current directory
- `cd directory` - change into another directory
- `rm file` - remove file (be careful because it won’t make a copy in the trash folder)
- `emacs` or `gvim` - text editors

You can find out more information about a command by typing: `man command` or `command --help`.
Image Reconstruction Software Project Page

Quick Links

All questions and comments should be directed to nadiamd@unimelb.edu.au

Package source code tar-ball - Release 0.1 (this is a test release with basic functionality).

Some preliminary specifications (written by Brian) - please give feedback on them!

Doxygen documentation of the source code (updated daily)

User questionnaire

Package Description

How to Get Started Easily

How to Install

How to Use