Standing Wave illumination of gold nanocrystals

Piotr Gryko
4th year MSci physics
Supervisor: Professor Ian Robinson

Abstract:

Coherent X-ray diffraction is a useful technique for understanding objects including those which can be represented as phase objects. X-rays are highly penetrating and have wavelengths very close to atomic spacing. Diffraction from a crystal lattice acquires phase whenever atoms are displaced from a lattice site allowing for strain fields to be mapped. Gold nanocrystals were imaged at the Advanced Photon Source producing a double beam diffraction image and simulations carried out to explain experimental diffraction patterns. Experimental images where then phased to produce a two dimensional real space image of the gold nanocrystals.

Contents

Introduction:
Many problems in science involve working on structures on the nanoscale, due to the interesting physical properties exhibited by materials in this size regime. Information regarding the size, density distribution, absolute free energy and structure of materials at this scale yield insights into the materials behaviour, promising a new class of designer materials. However, obtaining a three dimensional map of such a system is extremely difficult.
X-rays are useful for looking at such system, having a wavelength close to typical atomic spacings whilst being very penetrating. While imaging with x-rays is very appealing, there is a lack of suitable lenses available to obtain significant magnification. The result has been to abandon the use of a lens altogether, focusing instead on the use of computational methods, yielding a lens-less x-ray method called coherent x-ray diffraction (CXD).

The rest of this section provides background information for use in subsequent sections. A short description of the phasing problem in coherent x-ray diffraction and the results of my project will be outlined.

Background:
Illuminating a small object with a coherent X-ray beam results in a far field pattern, which when sampled in 2D, is related to the Fourier transform of the projection of the density of the object onto a plane perpendicular to the exiting diffracted wave vector. This simple relationship would allow for an inverse transform to image the object if the lost phase information could be recovered. This is the essence of coherent X-ray diffractive imaging.

We assume the crystal to be composed of unit cells arranged on a Bravais lattice, which is composed of an infinite number of points regularly spaced, so that the lattice appears identical when viewed from any lattice point. The fourier transform of a Bravais lattice is also a Bravais lattice, with the former referred to are as real space and the latter as reciprocal-space. A lattice vector R_n can be written as a linear combination of primitive lattice vectors a, b, c: $R_n = n_1 a + n_2 b + n_3 c$. The reciprocal space vector can be written as $H_n = ha^* + kb^* + lc^*$, with the primitive reciprocal lattice vectors determined by the real lattice vectors [2].
A crystal can be constructed by regularly repeating a basic structural unit, known as a unit cell. These may contain single atoms or large number of different atoms arranged arbitrarily. Hence a crystal can be thought of possessing a unit cell, convoluted with a Bravais lattice.
An X-ray photon can interact with an atom in two ways: it can be scattered or it can be absorbed. In the classical description of the scattering event the electric field of the incident X-ray exerts a force on the electronic charge, which then accelerates and radiates the scattered wave. Classically, the wavelength of the scattered wave is the same as that of the incident wave,

ensuring that the scattering is elastic. This is not true is a quantum mechanical description, where energy can be transferred to the electron, leaving the inelastic scattered photon with a lower frequency.
Elastic scattering is the one that is most exploited in X-ray imaging of materials, where the momentum of an incident photon ħ**k**, can be transferred elastically in a scattering event. The momentum transfer vector Q, can be defined as $\hbar Q = \hbar k - \hbar k'$, where ħ**k** and ħ**k**' are the initial and final momenta of the photon.
If we consider a collection of electrons to be a continuous charge density, ρ(r), the scattered radiation due to the distribution is the superposition of the radiated fields from each volume element ρ (r)dr at the point of observation P. By making the kinematical approximation, whereby we assume that each photon only interacts once with the charge distribution and the incident wave is a plane wave, the measured intensity can be written as [2]:

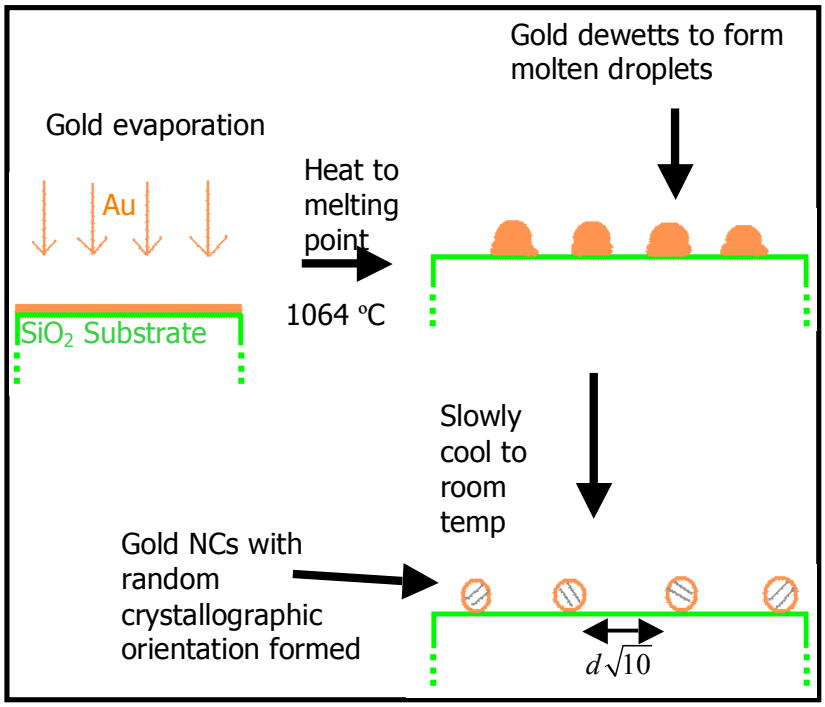
$$I(Q) = A^*(Q)A(Q) = \left|\frac{C}{V}\right|^2 \left|F\left(h_n\right)\right|^2 \left|\int s\left(r\right)e^{iQ.r}dr\right|^2 \qquad (1.1)$$

Where A(Q) is the diffracted amplitude, Q is the wavevector, V is the volume of the unit cell and F(h) is the form factor of the unit cell. The shape function s(r) can be a complex function $s(r) = \left|s(r)\right|e^{i\phi}$, with the phase angle ϕ containing information about the strain within the crystal.
The CXD pattern is collected by means of a 2D CCD detector, with the detector plane oriented perpendicular to the scattering vector K_f. The x and y axes are chosen to lie in the plane of the detector and z in the direction of K_f. Therefore a 2D slice though a CXD pattern is related to the Fourier transform of the projection of the read-space density onto the x-y plane.

Experimental Work:
Experimental work was carried out at the 34IDC beamline at the Advanced Photon Source (APS) in Argon (near Chicago). The APS is a third generation x-ray source using a synchrotron to generate high energy coherent x-rays. A cathode is used to produce 100 keV electrons, which are then accelerated via a linear accelerator to an energy of 450 MeV before being transferred to a booster ring. This then imparts energy of 32 keV per orbit until the desired energy of 7 GeV is reached. The 7 GeV electrons are then injected into the storage ring, which has a circumference of 1104m. The APS storage ring will normally hold about 100 mA of current. X-rays from the APS, of wavelength 1.38Å were used to image gold nanocrystals.

Gold Nanocrystal Synthesis
The synthesis of gold nanocrystals involved the evaporation of gold onto a silicon dioxide substrate, forming a 20nm polycrystalline layer. This was then heated close to the melting point of gold (1064ºC), causing it to de-wet from the surface of the substrate and form small molten droplets. These where then allowed to cool slowly to room temperature over a period of 12hrs, giving plenty of time for gold nanocrystals to from (see Fig 1). An empirical ratio of 1:10 exists for the deposition thickness to NC size. Hence nanocrystals in the range of 100 to 200nm where formed, with a random crystallographic

Fig 1: Gold NC synthesis



orientation with respect to the substrate. The separation between the nanoparticles is approximately $d\sqrt{10}$, where d is the diameter of the NCs. Hence for a deposition of 20nm, the nearest neighbour distance for 200nm NCs is approximately 600nm.

Experimental diffraction results:

The Ewald construction can be used to visualise the collected of CXD pattern, with figure 2 illustrating a 2D case.

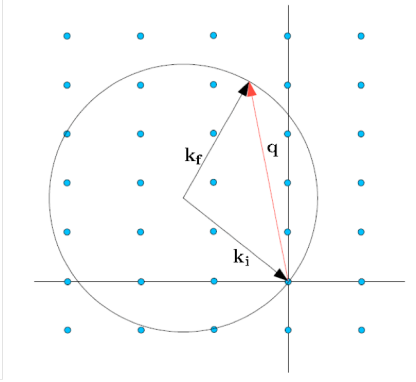


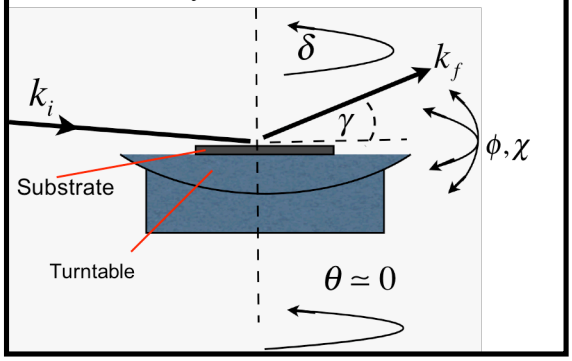
Fig 2 [2]: The Ewald construction can be used to visualise the collection of CXD data. The sample is rocked slightly in θ, rotating the diffraction triangle relative to the reciprocal lattice and thereby sweeping through the Bragg points.

Each point displayed in Fig 2 is a Bragg point. k_i is the wavevector of the incident radiation and points to the origin of reciprocal space. Neglecting inelastic scattering, allows the tail of the diffracted wave vector k_f to be coincident with the tail of k_i with its direction sweeping out a sphere. The diffraction triangle formed by k_i, k_f and q allows for all the Bragg points that lie on the sphere to be measured by moving the detector to the appropriate location on the sphere. Moving the plane of the detector along k_f creates a method for collecting data, where the 2D detector is represented by a 1D line in this case. The detector, which is always perpendicular to k_f, is moved through a Bragg point by very slightly rocking the incident angle thereby changing the direction of k_i. The sampled planes will then be out of parallel by a factor of $\sin\Delta\theta$.

In order to determine where to place the detector, a change of coordinates from the laboratory frame, where k_i and k_f are naturally defined, to the sample frame is necessary. This is normally done in the convention put forward by

Busing and Levy[18] and later extended to allow the experimenter to control the incident and exit angles at the sample[75].

Fig 3: Illustration of diffraction angles in laboratory frame of reference



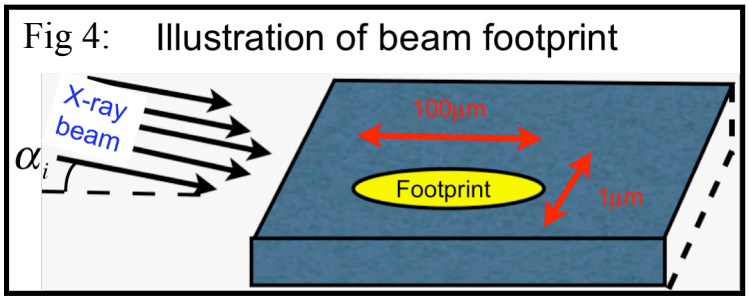
In the present experiment, a 4-circle diffractometer was used. The relevant angles are: 2θ, θ, φ, and χ. χ is a rotation about ki as illustrated in Fig 3. θ is the incident angle. φ is a rotation about the sample normal, 2θ is a compound angle made up of δ and γ and is not the same as θ. A further two angles are provided by the goniometer head and are used to align the sample so that it is perfectly flat with respect to the axis. 2θ is in the direction of θ and is constrained by q in the following way:

$$|q| = 2\frac{2\pi}{\lambda}\sin\frac{2\theta}{2} \qquad (1.2)$$

A combination of 2θ, θ, and χ was used to position the detector near a [111] Au Bragg point, with ϕ used to allow the selection of individual crystals from among those illuminated.

The 1.38 Å beam was prepared with a double crystal Si(111) monochromator, without focusing. The incidence angle of the beam was below the critical angle of the substrate (α_c~0.2ºC). Multiple accumulations of the diffraction pattern were taken to reduce the saturation of the CCD. Unlike forward scattering CXD, small angle x-ray scattering does not require a beamstop as the CCD is never exposed to the beam directly; only the reflected and diffracted part of the beam is imaged.

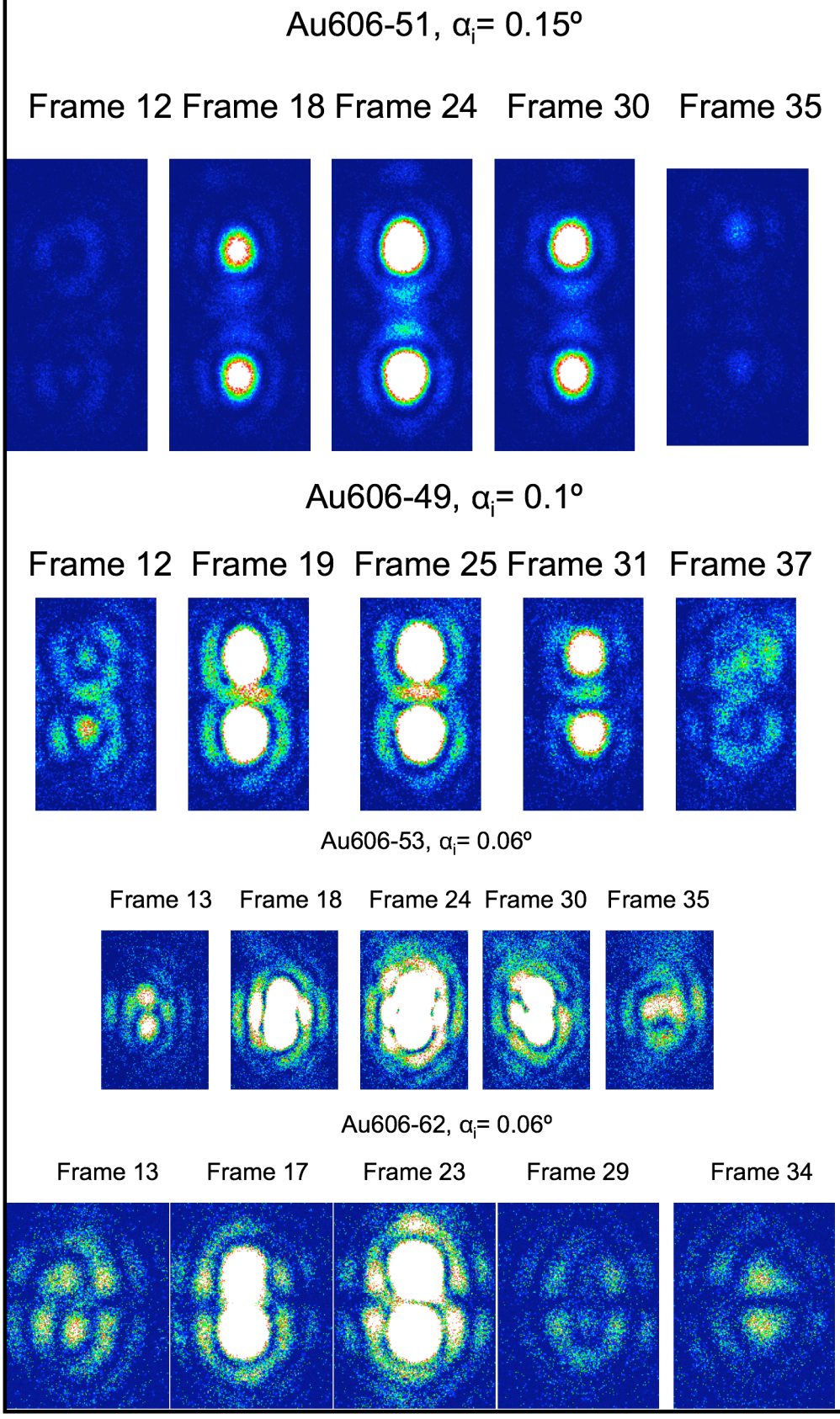
Fig 4: Illustration of beam footprint



Placing the gold NC coated substrate described above in the X-ray beam, finding a crystal and moving the detector to a [111] Bragg point yielded a double peak intensity distributions as shown in Fig 5. The beam diameter was 1µm, leaving an oval optical footprint of approximately 100µm by 1µm on the sample (see Fig 4). For 200nm gold NCs with a mean spacing of 600nm, the beam will illuminate approximately 250 NCs. All the NCs have a different crystallographic orientation with respect to the substrate and the CCD detector is able to detect 1° of the diffraction cone and 1° of the rocking curve. Therefore the probability of seeing one illuminated NC is $\frac{1}{(360)^2}$, with the

probability of seeing one gold NC when 250 are illuminated corresponding to $\frac{250}{(360)^2} \approx 0.002$. This means that despite the illumination of many gold NCs, it is rare to get the diffraction pattern for more than one Gold NC at the same time. Imaging two NCs with similar crystallographic orientations at the same time, would introduce a speckle pattern caused by interference between their corresponding diffraction patterns and hence could be differentiated from only seeing one NC. Images Au606_53, Au606_51, Au606_52 are diffraction patterns taken from the same nanocrystal but with different incidence angles α_i. As α_i is increased the separation of the primary peaks increases. The diffraction images produced show two central peaks surrounded by circular fringe, which exhibits 3-fold modulation. At each incidence angle, the NC were rocked in θ, which causes the 111 peak to pass through the diffraction sphere (ie the Ewald sphere) producing a series of image frames as illustrated in fig 5. Another NC was imaged (Au606_62) at $\alpha_i = 0.06$ producing a diffraction pattern with similar features to the previous NC. However there is a central area of cancellation between the two central peaks in Au606_62 compared to a maxima in Au606_49. Since the intensity distribution is related to the square of the Fourier transform of the crystal shape, we can identify the prominent features of the patterns. The diffraction pattern resembles two copies of the Bessel function, which is the FT of a sphere.
File Au606_62 was chosen for further close analysis, due to its unusual central cancellation, the closeness of its Bragg peaks and its symmetry.

Fig 5: Two different nanocrystal diffraction images, recorded at different incidence angles. The first NC is saved as Au606-53, Au606-49, Au606-51 and the other NC as Au606-62



Standing wave illumination

To support the experimental work, a theoretical model of the system was created to simulate the diffraction pattern. Simulations where written in C, compiled using GCC and called using python scripts [10]. The NC is illuminated with a beam of coherence length greater than that of the object, allowing for the beam to be treated as being coherent.
The incident beam reflects off the substrate, at an angle bellow the critical angle of α_c, which was recorded as 0.2 during the experiment. The two incident waves K_1 and K_2 interfere with each other creating a standing wave which illuminated the nanocrystal object (Fig 6).

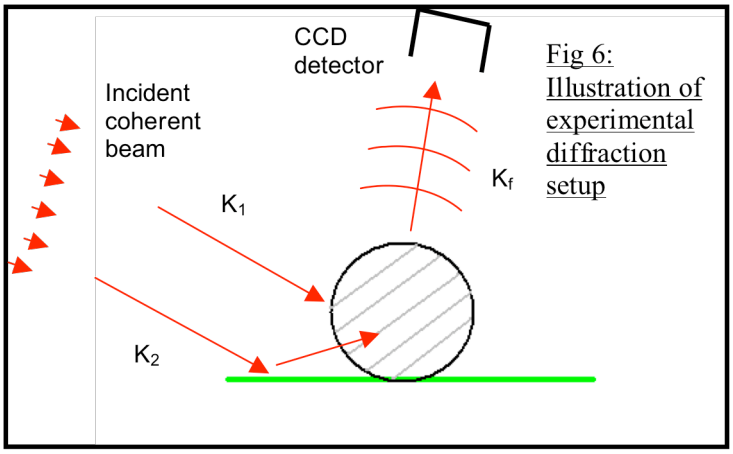


Fig 6: Illustration of experimental diffraction setup

This can be represented as two incident waves with a difference of ΔK between a reference vector k_0 (Fig 4).

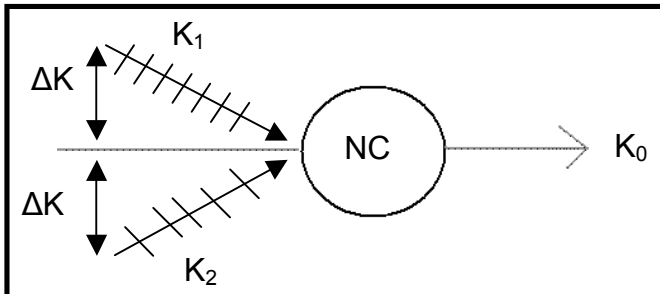


Fig 7: Theoretical basis of diffraction model Two incident waves k_1 K_2 diffract off the NC, with a difference of ΔK between a reference vector K_0

For simplicity the incident wave vectors (k_1 and k_2) are confined to the x-y plane, letting:

$$\vec{k_1} = k_0\hat{x} - \Delta k\hat{y}$$
$$\vec{k_2} = k_0\hat{x} + \Delta k\hat{y}$$

Giving the total amplitude as: $A_{Tot} = A_1 e^{i(k_0\hat{x}-\Delta k\hat{y}).\vec{r}} + A_2 e^{i(k_0\hat{x}+\Delta k\hat{y}).\vec{r}}$

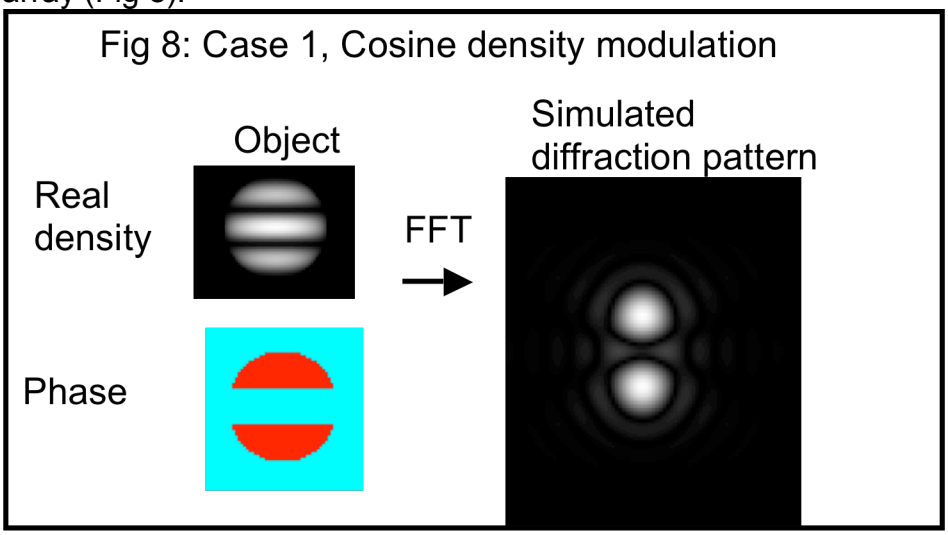
Three separate cases where considered in the simulation and compared with experimental data:

1) For both incident waves being the same (A1 = A2), the total amplitude A_{Tot} can be represented as containing a cosine modulation:

$$A_{Tot} = A_1 e^{ik_0\hat{x}.\vec{r}}(e^{-i\Delta k\hat{y}.\vec{r}} + e^{i\Delta k\hat{y}.\vec{r}}) = 2A_1 e^{ik_0\hat{x}.\vec{r}} Cos(\Delta k\hat{y}.\vec{r}) \qquad (1.3)$$

With x.r and y.r being represented as the x and y position across the array in the simulation. Hence the particle was modelled as a circle of radius 20 pixels, with its density (z-axis) modulated by a cosine function. To represent the superposition of two incident beams, the density was multiplied by Cos(k.y), with K representing the incident angle and y the vertical distance across the array (Fig 8).

Fig 8: Case 1, Cosine density modulation



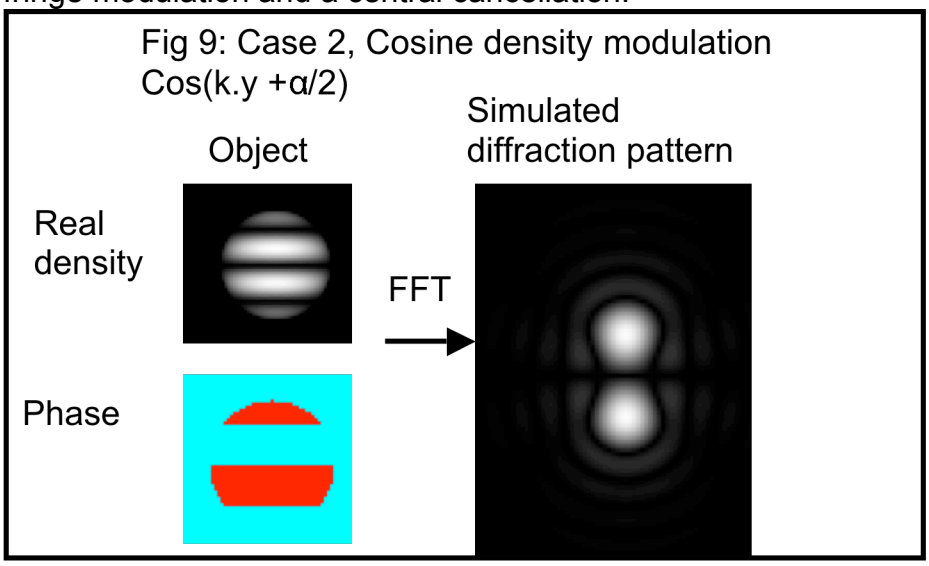
In this case the wave A_2 has a standing wave setup in only the y-direction. Increasing the value of K increases the separation between the peaks in the FFT image. The wavelength of the standing wave is 2π/ΔK, with delta K being quite small as the waves are almost parallel.

Figure 8 illustrates this process. The first stage involves creating a 3d complex array of size 256 by 256 pixels, containing x and y coordinates with an associated complex density. The object is then drawn as circle of points with unit density, placed on an x-y plane; the circle at this point only consists of a real part. The next stage is to apply the cosine function to the real part of the density creating a modulation in the y axis. This is visible in the above illustration, where the real object density is shown to be a circle with horizontal stripes. The image bellow it shows the phase ($e^{i\phi}$) of the object. The phase is a cyclic function going from red, which represents $e^{i0}=1$, to green ($e^{\frac{i\pi}{2}}$), to light blue ($e^{i\pi}=-1$), to dark blue ($e^{\frac{i3\pi}{4}}$) and then back to red. Hence for this simulation the object is entirely real, containing only red and light blue colours for phase. The final stage is to Fourier transform the object to produce the simulation of the diffraction pattern. The simulation diffraction pattern contains features found in the experimental data, such as two primary peaks and a similar outer fringe shape. However the central cancellation is not as proficient and the fine fringes do not decay at a fast enough rate.

2) For the case where there is a phase difference between waves A1 and A2, i.e $A_2 = A_1 e^{i\alpha}$

$$A_{Tot} = A_1 e^{ik_0 \hat{x}.\bar{r}}(e^{-i\Delta k \hat{y}.\bar{r}} + e^{i(\Delta k \hat{y}.\bar{r}+\alpha)}) = A_1 e^{(ik_0 \hat{x}.\bar{r}+\alpha/2)}(e^{-i(\Delta k \hat{y}.\bar{r}+\alpha/2)} + e^{i(\Delta k \hat{y}.\bar{r}+\alpha/2)})$$
$$= 2A_1 e^{i(k_0 \hat{x}.\bar{r}+\alpha/2)} Cos(\Delta k \hat{y}.\bar{r} + \alpha/2) \qquad (1.4)$$

This leads to two variables in the cosine term, K the standing wave wave-vector and α the phase difference.
The phase difference factor α, corresponds to an optical path difference introduced into the reflected wave A_2 and hence modifying α changes the position of the standing wave across the NC. The structure of the NC substrate interface is unknown, creating an uncertainty regarding the distance of the nanoparticle from the substrate. A change in the NC height with respect to the substrate, introduces an optical path difference, which can be absorbed into the phase factor α. It is also possible to change α by varying the incidence angle, thereby changing how the NC is illuminated by the standing wave. An asymmetry in the standing wave illumination may help in the phasing of the diffraction image.
Hence the particle can now be modelled as a circle with its density multiplied by cos(k.y + α), with K representing the standing wave wave-vector and α the additional phase factor (Fig 9). Increasing the value of K increases the separation between the primary peaks in the FFT images, whereas increasing α cyclically shifts the central minima into a maxima. A value of 0.24 pixels for K and 1.5 pixels for α, was found to closely match experimental data (frame 23 Au606_62), with the simulated diffraction image having a more defined fringe modulation and a central cancellation.



3) For the case where both the amplitude and the phase of the waves are different ($A_1 \neq A_2$) but still similar ($A_2 \approx A_1 e^{i\alpha}$), let $A_2 = A_1 \beta e^{i\alpha}$

$$A_{Tot} = A_1 e^{(ik_0 \hat{x}.\vec{r} + \alpha/2)} (e^{-i(\Delta k \hat{y}.\vec{r} + \alpha/2)} + \beta e^{i(\Delta k \hat{y}.\vec{r} + \alpha/2)})$$
$$= 2A_1 e^{i(k_0 \hat{x}.\vec{r} + \alpha/2)} [(\beta + 1) Cos(\Delta k \hat{y}.\vec{r} + \alpha/2) + i(\beta - 1) Sin(\Delta k \hat{y}.\vec{r} + \alpha/2)] \quad (1.5)$$

Where attenuation coefficient β, lies in the range $0 \leq \beta \leq 1$ and represents any loss in intensity of the reflected beam due to absorption and scattering.

This requires a change in the simulation, removing the cosine modulation to the real part of the density and setting the real part to equal: density*cos(K*y*0.01 + Phi*0.1)*(B +1), and the imaginary part to equal: density*sin(K*y*0.01 + Phi*0.1)*(B -1)

The absorption parameter B creates an asymmetry in the diffraction image, causing one of the central peaks to appear brighter than the other (Fig 10). This effect can be seen in the experimental data but is modest in magnitude. A calibration of the experimental and theoretical image data in origin yielded B as being 0.96 using frame 23 of Au606_62 as a reference.

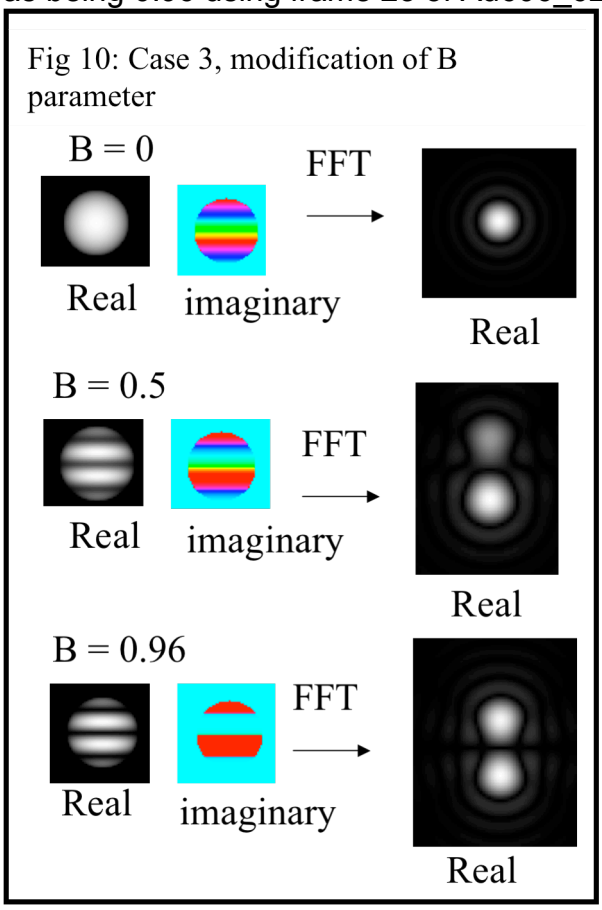


Fig 10: Case 3, modification of B parameter

Reduction of secondary fringes

It was still observed that secondary fringes where not decaying fast enough compared to experimental data. This was attributed to sharp density drop on the sides of the simulated sphere where the density was going from 1 to zero over the space of a pixel. A Fermi function was chosen to introduce soft edges on the circle due to its ease of implementation, allowing for a change in density from 1 to 0 over some distance (Fig 11). The Fermi scaling function was tested in Mathematica in 2d and then applied to both the x and y direction (Fig 12).

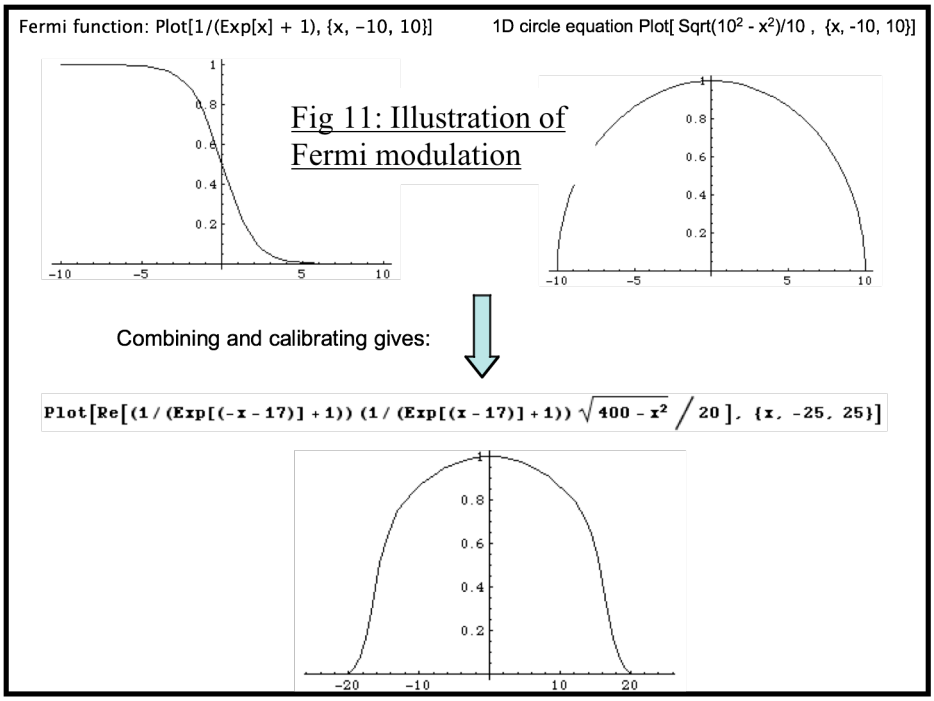


Fig 11: Illustration of Fermi modulation

When applied to the simulation, the resultant FFT gained an improvement in the modulation of the primary fringe and an increase in the decay of the later fringes.

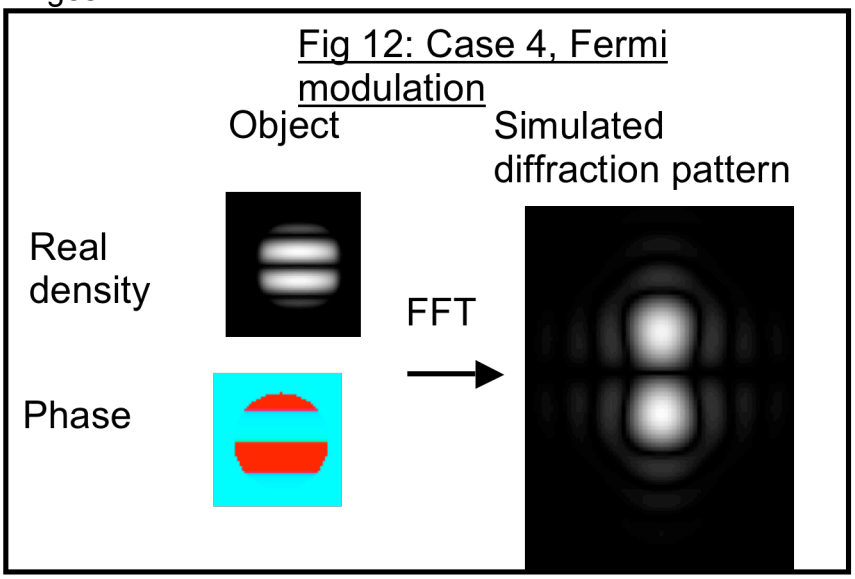


Fig 12: Case 4, Fermi modulation

Comparison of theoretical and experimental data:

An object is related to its diffraction pattern via means of a Fourier transform, which is an integral from minus to plus infinity and is therefore a continuous function. Computers however store data, such as the diffraction images in discrete format or finite length and require discrete Fourier transforms (DFT) to be used.

FFT and DFT

The time and frequency domains are alternative ways of representing signals. The Fourier transform is the mathematical relationship between these two representations. If a signal is modified in one domain, it will also be changed in the other domain, although usually not in the same way. The Fourier transform of a one dimensional object f(x) is defined by [1]:

$$F(q) = \int_{-\infty}^{\infty} f(x) e^{iqx} dx \qquad (1.6)$$

With the inverse being:

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(q) e^{-iqx} dq \qquad (1.7)$$

A Fourier transform is continuous function, but as digital computers can only work with information that is discrete and finite in length, discrete Fourier transforms (DFT) are used. DFTs view both the time domain and the frequency domain as periodic

$$F(n) = DFT(f(m)) = \frac{1}{N} \sum_{m=1}^{N} f(m) e^{2\pi inm/N} \qquad (1.8)$$

Where F is a real-space function, f is its Fourier transform and the discrete array occupies N points. The inverse transform is:

$$f(m) = DFT^{-1}(F(n)) = \sum_{m=1}^{N} f(n) e^{-2\pi inm/N} \qquad (1.9)$$

Parsivals theorem connects the two arrays, ensuring that the integrated 'energy' in the two discrete arrays is the same:

$$\sum_{n=1}^{N} \left|F(n)\right|^2 = \frac{1}{N} \sum_{m=1}^{N} \left|f(m)\right|^2 \qquad (2.0)$$

Scaling of simulated data

The simulated data array sizes need to be rescaled for a direct comparison with experimental data. The intensity of the measured diffraction image , is related to the Fourier transform of the of the object density:

$$I(Q) = \left|F(Q)\right|^2 \text{ and } F(Q) = \int_{-\infty}^{+\infty} \rho_{eff}(\vec{r}) e^{i\vec{Q}.\vec{r}} d^3\vec{r}\ , \qquad (2.1)$$

Where ρ_{eff} is the effective electron density, $\vec{Q}$ is the momentum transfer and $\vec{r}$ is the objects position vector. The DFT performed in the computer simulation in 1D is related to the FFT via the following relationship:

$$\int_{-\infty}^{+\infty} \rho_{eff}(x) e^{iQ.x} dx \xrightarrow{\text{Discrete}} \sum_{j=0}^{N-1} \rho_j e^{i\frac{2\pi jk}{N}} \qquad (2.2)$$

And by equating exponentials $Qx = \frac{2\pi jk}{N}$, where j and k are indexes, and N is the number of points in the array. The experimental diffraction images are sampled and stored in arrays of size N as illustrated in Fig 13.

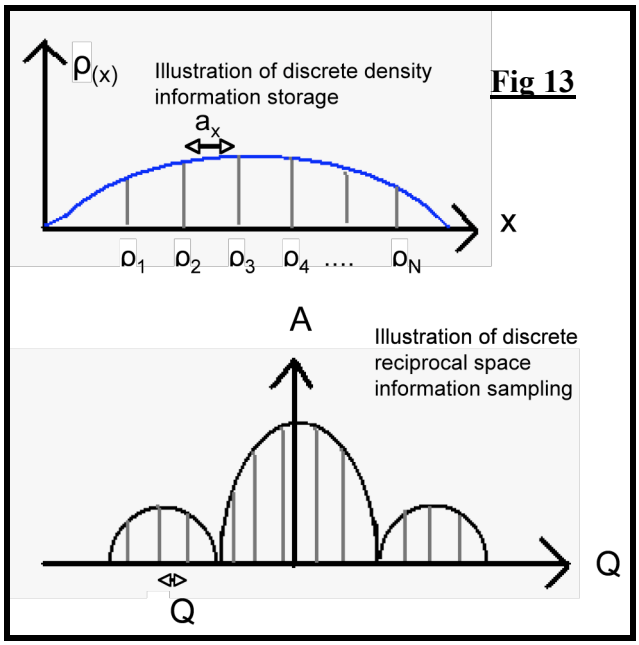


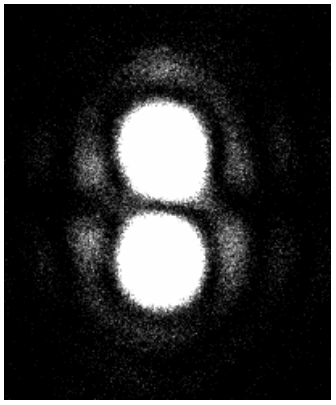
Fig 13

Each density measurement ρ_j is carried out at a regular sampling interval a_x, with j being an integer. A similar process is carried out with reciprocal space data, hence we can write $Q = k\Delta Q$ and $x = ja_x$ where k is an integer and ΔQ is the sample spacing for Q. This allows for the relationship between the Fourier transform and the DFT to be written as:

$$Qx = k\Delta Q.ja_x = \frac{2\pi jk}{N}$$

Cancelling the indices j and k leaves $\Delta Q.a_x = \frac{2\pi}{N}$ which relates the sampling size in Q space and real space to the size of the array. From the grating equation, ΔQ can be derived from $\Delta Q = \frac{2\pi p}{\lambda D}$, where p is the CCD pixel size, λ is the x-ray wavelength and D is the distance from the sample to the CCD. From this formula ΔQ was calculated to be 1.82x106m-1 (λ = 0.138nm, p = 40μm and D = 1m).

Comparison of theoretical and experimental data cross-sections using origin
The simulated diffraction pattern was modelled to closely resemble images from Au606-62, particularly frames 23 and 24 as these where found nearest to the bragg peak. These where imported into origin for analysis (Fig14), with horizontal and vertical cross-sections taken, square rooted to give the amplitude and compared with similar cross-sections with simulated data.



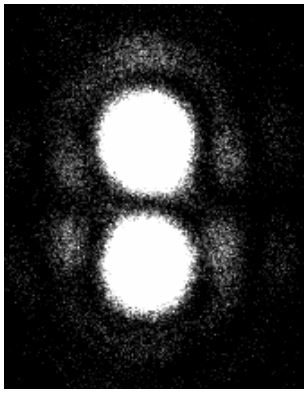


Fig 14:
Frames 23 and 24
From Au606-62

Frame 23 Frame 24

Rescaling of simulated data

Cross-sections of data where taken from the experimental diffraction images and the simulated diffraction images and compared with each other. Specifically frame 23 from Au606_62 was found to be most symmetric and hence was used as a reference. The scaling factor between the two images corresponds the ratio between the ΔQ_{exp} and ΔQ_{sim}, which was found to be 3.5:1. Through the relationship $\Delta Q.a_x = \frac{2\pi}{N}$, the simulated array size N was changed from 256x256 pixels to 896x896 pixels, thereby rescaling ΔQ spacing of the simulated diffraction images so that they matched the experimental ones. The simulated data was centered and the intensity rescaled to that of Au606_62 (Fig 15).

Fig 15

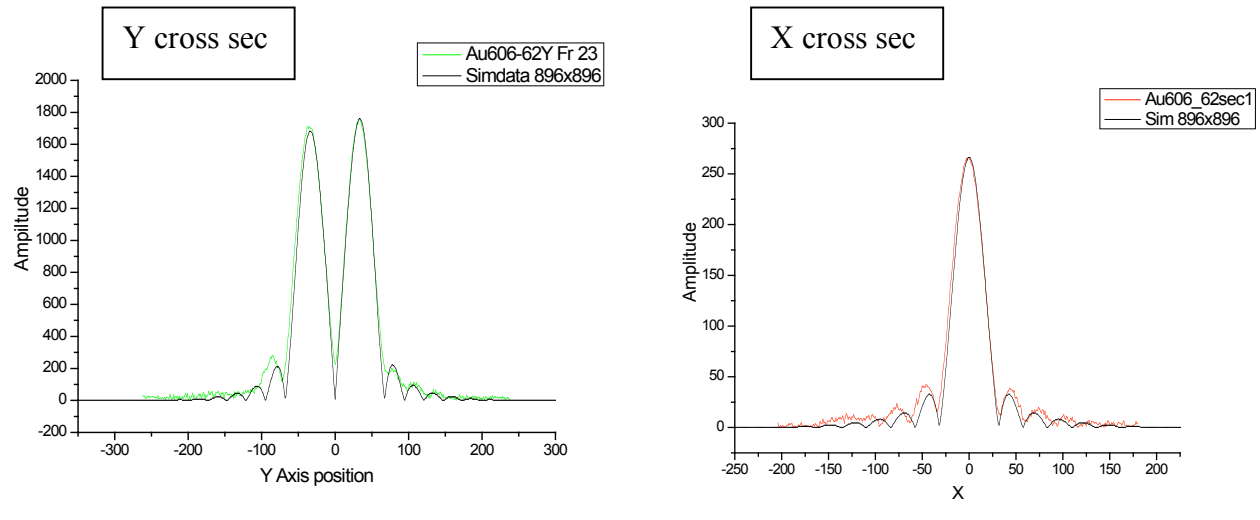


Fig 15 consists of a comparison of frame 23 of Au606-62 (red and green lines) with the simulated diffraction image (black lines). The parameters of the simulation have been chosen to give a close as possible fit to experimental data. As mentioned in the simulation section parameters of 0.24 pixels for K, 1.5 pixels for α and 0.96 for B where used. There is a good agreement, with both graphs closely matching the experimental data. There is an asymmetry

in the left side of the Y cross-section where the two sets of data diverge and there is a small offset in the horizontal scaling of the x cross-section. The same ΔQ scaling (3.5) was used for both the x and y axis for the simulation. However there is a slight difference between the x and y scaling, with the correct x scaling corresponding to 3.7. This is likely due to the NC not being perfectly circular in shape and is more likely to be slightly elliptical as suggested by the different ΔQ scalings. The NC size can then be calculated from $\Delta Q.a_x = \frac{2\pi}{N}$, with (ΔQ 1.82x106m-1) thereby giving a value of a_x = 170 ±20nm.

Phasing

Phasing algorithms

Illuminating a small object with a coherent X-ray beam results in a far field pattern, which when sampled in 2D, is related to the Fourier transform of the projection of the density of the object onto a plane perpendicular to the exiting diffracted wave vector. The scattered amplitude at a point is a complex quantity $A(p) = |A(p)|e^{i\phi(p)}$ of which only the intensity i.e. $A^*(p)A(p) = |A(p)|^2$ is known, leading to a loss of all phase information in the scattered wave field at P.

This simple relationship would allow for an inverse transform to image the object if the lost phase information could be recovered. This is the essence of coherent X-ray diffractive imaging [2]. Methods have been developed for microscopy via X-ray holography and through the application of iterative computational methods, which are used as part of this project.

The phasing algorithms used, function by successively applying real and reciprocal space constraints to an iterate, with the earliest phasing algorithm developed by Gerchberg and Saxton in 1972[7]. Iterative phase retrieval works because generally there are very few sets of phases [3,4,5,6] that can be matched to the measured amplitudes when a compact support is used for the real-space function. Two algorithms where used, the Error Reduction which was developed by Fienup [8] based on Gerchberg and Saxtons work, and the hybrid input output.

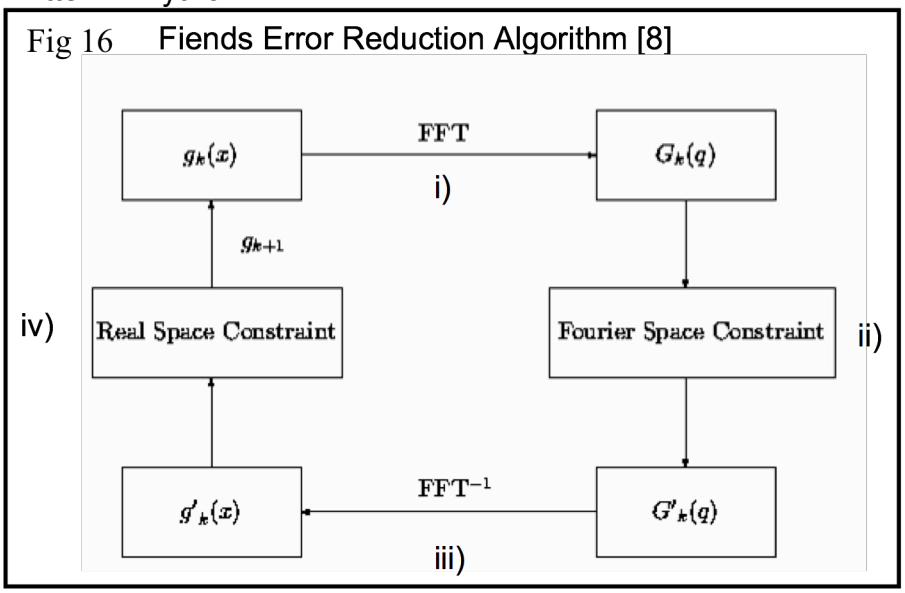
An implementation of the ER algorithm would involve the following:

i) Fourier transform an estimate of the real space density gk
ii) Make the smallest change possible to Gk, the Fourier transform of gk, to satisfy the Fourier modulus constraint,
iii) Back transform the resulting estimate, Gk of the actual diffracted amplitude.
iv) Make the smallest changes possible to the calculated real-space density, g_k so that it obeys the real-space constraint to arrive at an estimate of the real-space density g. This cycle is demonstrated schematically in Fig 16.

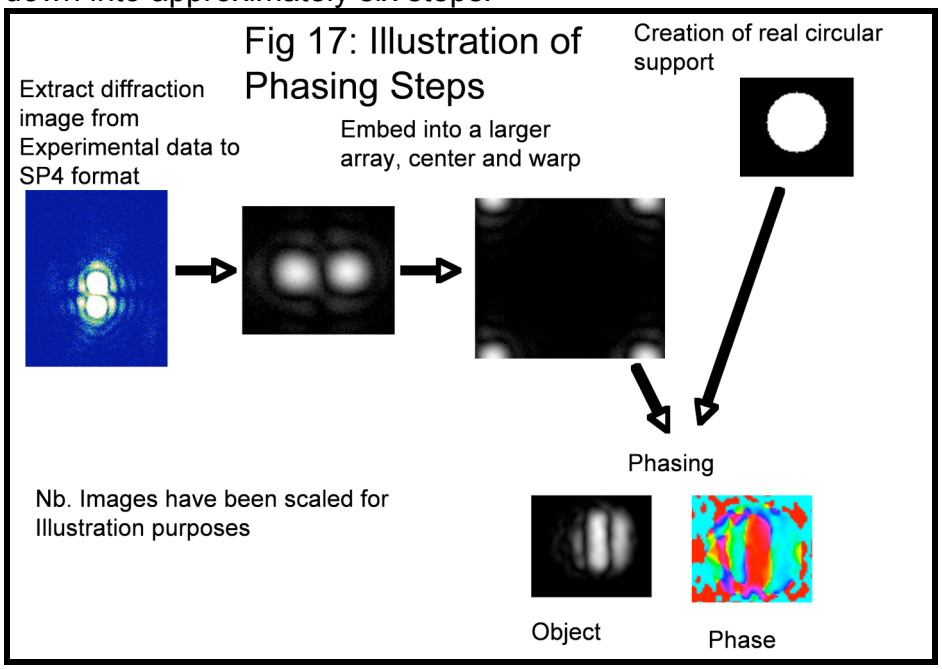
The Hybrid Input Output method is a modification of the ER, relying on a sufficiently tight support constraint without positivity to allow for the reconstruction of a complex valued real-space density.

Phasing steps:
Phasing functions and algorithms are used to directly recover the phases of diffraction amplitudes from the recorded intensities. The data is presumed to be stored in one, two or three dimensional arrays of the Sp4Array format. They are in the form of functions written in C, meant to be run from scripts written in Python.

Fig 16 Fiends Error Reduction Algorithm [8]



The phasing of Experimental data (illustrated in Fig 17) can hence be broken down into approximately six steps:



1. Firstly the experimental diffraction images need to have background noise subtracted using reference background diffraction data.

2. The required diffraction images then need to be cut out and converted from their native .SPE format into an .SP4 format. A python script called speConvert.py was used for this purpose (see Appendix).
3. The .SP4 file is then centred (Centre.py, see Appendix) and Embedded in larger .SP4 array (Embedarray.py, see Appendix). Centering is important for phasing, as any offset results in an additional phase shift in the phased object. The intensity of the imbedded diffraction pattern is square rooted to give the necessary amplitudes, with which the phasing algorithms can be used. Centering is achieved by embedding the diffraction image in a larger array, at an estimate of its centre and then applying an FFT. This gives the autocorrelation function of the diffraction image, whose phases change depending on centering of the diffraction image . For a correctly aligned diffraction image the autocorrelation functions phases are flat (i.e. constant), hence an iterative script was used to generate multiple autocorrelation images at slightly different staring values.
4. The centred and imbedded diffraction images then need to be wrapped to make the phases sensible in the Fourier Transform. The wrapping is done by a forward FFT and multiplying the resulting complex number in element [i,j] by (-1)^(i+j) then FFTing back. If this is not done the phases after FFT will have a (-1)^(i+j) modulation, where i,j are the array indicies. This process is included in the Embedarray.py script.
5. A support needs to be created for the object. This acts as a real space constraint for the phasing algorithm, indicating the allowed physical size of the phased object. The support must be made slightly larger than the expected phased object. A large support offers more possible solutions for the phasing algorithms, resulting in a lower quality resultant object. A too small support, results in the object being eaten away at, producing a phased object with the same shape and size as the support. Hence the goal is to find a tight support, which is as close in size to imaged object as possible (See Fig 18). Two C programs where written for this purpose; 2dcirclesupport.c creates a circular support at a specified user location in the array and 2dpolysupport.c creates polygon support using a configuration file to specify the support polygons dimensions.

Fig 19: Progression of phasing of Au606-62 Frame 23 after each ER-HIO-ER cycle

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

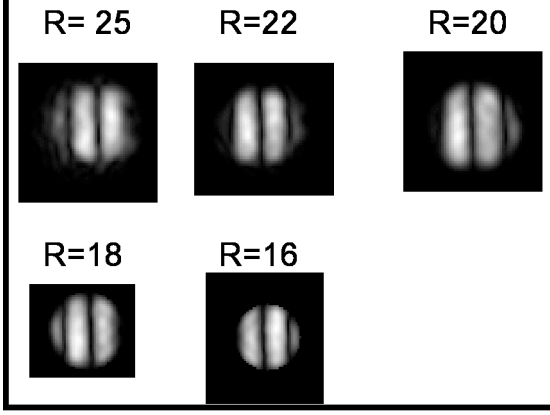
The phased NC will move around its support during the phasing cycle, usually displaying a gradual convergence to a resulting image.

6. Phasing is the carried out using a script called Phase.py (see appendix). This loads the centred-wrapped-square rooted experimental data and the support file. The Error reduction algorithm is run for 100

iteration, then the hybrid input output for 500 cycles and finally 100 ER iterations. This process of ER, HIO and ER is then run five more times, at each stage outputting a resulting real space image. At each cycle of ER-HIO-ER the phased object tends to converge more until it reaches an equilibrium state (see Fig 19), which depends on the support used and the object being phased.

Fig 18 Reduction of support size

Frame 23 of Au606-62 was phased using circular supports of different radii. The quality of the phased image increases as the support becomes closer in size to the actual object. However, once the support becomes too small, it begins to eat into at the object leaving a phased object of a similar size and shape as the support

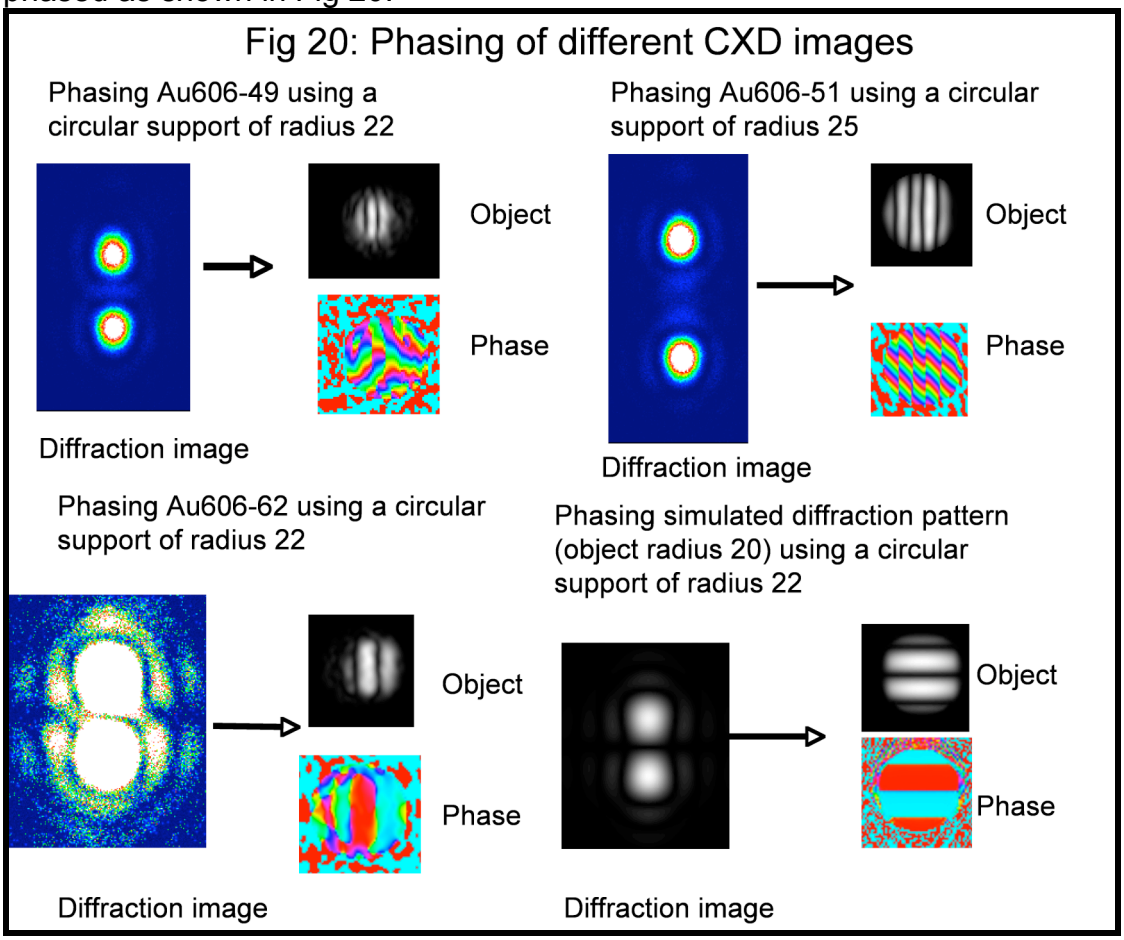


Phasing using a simulated object involves a similar process:
1) A real space double diffracted object is simulated (2dcircledisloc.c, see Appendix) as outlined in the previous section and saved as an .SP4 file. The simulation was written in C and compiled using GCC. The function requires a configuration file containing the 2d array size, the object size and position. Additional parameters such as the phase factor (α) and standing wave wave-vector (K) are entered when the program is called. Hence the program is run by typing "2dcircledisloc9 <ConfigFileName> <OutputFileName> <KValue> <BValue>".
2) The object then needs to wrapped and Fast Fourier Transformed to produce an .SP4 data file (sqrtwrappeddata.SP4), which will be fed into the phasing scripts. If the object is wrapped again after the FFT, a simulated diffraction amplitude image is produced, which can be compared with the square rooted experimental data.
3) A support is created in an identical way as for the experimental data. As the array sizes have been calibrated, an identical support as that for experimental image Au606-62 can be used.
4) Phasing is then carried out using a scrip called Phase.py. This runs loads the centred-wrapped-square rooted experimental data and the support file. The Error reduction algorithm is run for 100 iteration, then the hybrid input output for 500 cycles and finally 100 ER iterations. This process of ER, HIO

and ER is then run five more times, at each stage outputting a resulting real space image.

Experimental data from files Au606_49, Au606_51 and Au606_62 where phased as shown in Fig 20.

Fig 20: Phasing of different CXD images

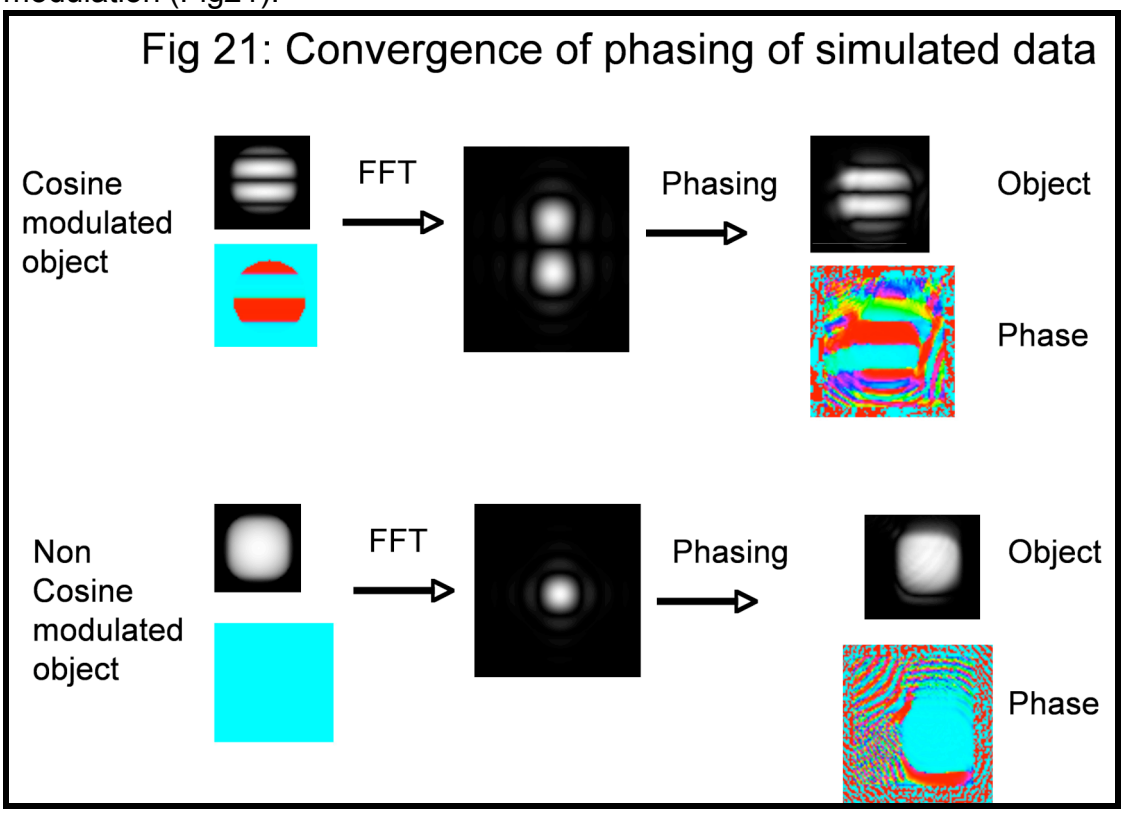


All three of the experimental diffraction images and the simulated image, where successfully phased. Comparing Au606-62 with the simulated diffraction pattern, the results are very similar. Both their real space objects display a circle with a cosine modulation, with the diffraction image showing a greater asymmetry than the simulated image. The phases are also very similar with both having a red region (corresponding to $e^{i0} = 1$) near the centre and near the edge, with the experimental image again displaying an asymmetry. The phased simulated images are rotated 90° with respect to the experimental images due a formatting error.
The phased images from Au606-49 and Au606-51 both share a cosine modulation and circular shape as seen in Au606-62, however the period of modulation is greater for Au606-49 and smaller for Au606-51, corresponding to a difference incidence angle. There is a strong striping in the phases, especially in Au606-51, as a result of Au606-49 and Au606-51 not being properly centred. The period of the striping is related to the magnitude of the diffraction pattern offset, with a greater offset resulting in a smaller period.

Convergence of Phasing

One of the founding ideas behind this project was the hope that the standing wave illumination would aid the phasing algorithms in the conversion of experimental diffraction data. X-rays have indices of refraction very slightly less than unity in most materials [1,9] making the manufacture of focusing optics very difficult, leaving Phasing algorithms as a possible alternative. However phasing has many flaws, with the algorithms having difficulty working with symmetric objects or creating vortices and stagnating [1]. The introduction of asymmetry into a diffraction image by the introduction of a stand-wave should therefore aid phasing. To test this hypothesis a simulation was carried out, comparing a standing wave simulation (with circular radius 20 pixels) with another circular object (R=20) but minus the standing wave modulation (Fig21).

Fig 21: Convergence of phasing of simulated data



A large square of 70 by 65 pixels support was chosen to make phasing difficult, with the hope of seeing an improved phasing result in the standing wave simulation. The result was inconclusive with both objects phasing with large supports of 70x65 and becoming incoherent at larger support sizes. There are several factors to take into account when looking at phasing convergence; the degree of asymmetry of the object could be effecting the success of phasing of standing wave objects, with there being a possible optimum asymmetry. In addition to this, the simulated objects always phase more quickly and are able to tolerate larger support sizes than experimental images, largely due to the absence of noise and beam coherence effects found in experimental data. In a CXD experiment, the x-ray beam has to pass through multiple optics, introducing an additional complexity to the x-ray beam [1]. Hence any benefit from using the standing wave illumination cannot be easily determined from simulations.

Conclusion and Summery

Experimental CXD data taken from a summer project was used as a basis for the 4C00 project building on the work previously carried out in several ways:

- The experimental data was analysed in Origin
- This allowed for a theoretical simulation to be built, which was finely tuned to closely match experimental results.
- Both the simulated and experimental data was then phased, proving that the algorithms developed for forward CXD, can cope with the standing wave illumination.
- The convergence of standing wave illumination was contrasted with non-standing wave, using simulated data. This yielded an inconclusive result.

Possible further work, could follow along the following directions:

- A more rigours method of analysing convergence could be developed, to quantitatively measure the rate and extent of convergence of a phased object. A chi-squared parameter is already used in the phasing algorithms and could be further investigated.
- Direct experimental observations could be carried out to prove whether phasing is easier using a standing wave illumination compared to forward scattering.
- An iterative support fitting program could be developed to generate optimum supports for phasing.
- The simulated diffraction model could be more finely tuned. As mentioned earlier the x and y scaling of the experimental data is different, suggesting that the object is elliptical rather than circular in shape. A refinement of the model could take this into account.
- Different types of wetting could be modelled. A Fermi function was used in the simulation due to its ease of implementation however the exact way in which the NCs sit on the substrate is unknown.

Finally the step could be taken to extend from simulating and imaging a 2D object to that of a 3 dimensional one.

References:

[1]Elements of modern x-ray physics, Jens Als-Nielsen, Des McMorrow Physics. Wiley, 2001.
[2]Microscopy of gold microcrystals by coherent x-ray diffractive imaging, Garth Williams, PhD Thesis
[3] Fourier phase problems are uniquely solvable in more than one dimension, R.H.T. Bates, Optik, 61:247–262, 1982.
[4] On the ambiguity of the image reconstruction problem, Yu.M. Bruck and L.G. Sodin, Optics Communications, 30:304–308, 1979.
[5] Unique reconstruction of a band limited multidimensional signal from its phase or magnitude, J. L. C. Sanz and T. S. Huang. J, Opt. Soc. Am., 73:1446–1450, 1983.
[6] Necessary conditions for a unique solution to two-dimensional phase recovery, R. Barakat and G. Newsam, J. Math. Phys., 25:3190–3193, 1984.
[7] A practical algorithm for the determination of phase from image and diffraction plane pictures, R.W. Gerchberg and W.O. Saxton, Optik, 35:237–246, 1972.
[8] Reconstruction of an object from the modulus of its Fourier transform, J.R. Fienup, Optics Letters, 3:27–29, 1978.
[9] X-ray Diffraction B.E. Warren Dover Publications, Inc., 1990.
[10] Dr Ross Rharder, University of Illinois

Appendix:

2dcircledisIco9.c
2dcirclesupport.c
center.py
embedarray.py
RealFFT.py
Phase.py
SpeConvert.py

```
/*  2dcircledisloc9.c

     Simulate a hemisphere, store in 2D array nn[2]ith unit amplitude.
     Add modulation of form cos(k.y + Phi) to density to simulate double beam
diffraction
     where k is wavevector and Phi is phase shift of reflected beam. Saved as sp4 file
     Add fermi scaling to smooth hemisphere
     This version removes useless code.
     Piotr 3/12/06

Requires a config file.
Correct config files are of form:

#array size
256 256
# x, y, radius
15 0 20

Run program using command:
2dcircledisloc8 conf image 24 15

(where 24 is wavevector (K) and 15 is phase shift (Phi)):

Images can be produced using commands:
2dcircledisloc8 conf image 24 15; python 2dplot.py image; python FFT.py image
*/

#define X_DIM 512
#define Y_DIM 512
#define N_DIM 2
#define N_PLANES 20
#define UNITY 40
#define DELTA 5
#include "sp4util.h"
#include "math.h"

int main(int argc, char* argv[]){

     int x,y,i,j,R,x0,y0,ramp,nt,K,Phi,radius,r2,N=0,ND=0,n,it=0,a0;
     double plane[N_PLANES][3];
     double disloc[N_PLANES][4];
     double ff,density,dd,xfermi,yfermi,B;
     FILE* fp_in;
     Sp4Array ar;
     unsigned long nn[4];
     char* comments = "IKR 2/29/04 make pictures of complex dislocation strain
fields";
     char str[25],name[25];             /* each input line must be less than 25 chars
long! */
     char* string1[20];
     ramp = 0;
```

```
     /* check for usage and read in parameters */
     if(argc!=5){
          printf("Usage:\n\t2dpoly <config_file> <output_file> <K_integer>
<Phi_integer>\n");
          return BAD_PARAMETERS;
     }

     if((fp_in=fopen(argv[1], "r")) == NULL){
          printf("2ddisloc:  Can't open %s\n", argv[1]);
          return LOAD_FAILED;
     }
     fgets(str, 25, fp_in);
     while(str[0]=='#') fgets(str,25,fp_in);
     sscanf(str,"%ld %ld",&nn[2],&nn[1]);

     sscanf(argv[3],"%ld",&K);
     printf("Wavevector is %ld\n", K);

     sscanf(argv[4],"%ld",&Phi);
     printf("Phase shift (Phi) is %ld\n", Phi);

     sscanf(argv[2],"%s", &string1);

     /* was originally trying to rename output files by merging strings
     sprintf(name,"%s%d",string1,B);
          printf("The file name is %s\n",name); */


     for(i=0;i<N_PLANES;i++){
          fgets(str, 25, fp_in);
          while(str[0]=='#') fgets(str,25,fp_in);
          if(str[0]=='\n' || str[0]=='\r') break;
          if(feof(fp_in) != 0) break;
          N++;
          plane[i][1] = strtod(strtok(str, " \t"), NULL);
          printf("Centre of Circle %d: X=%5.1f ", i, plane[i][1]);
          plane[i][2] = strtod(strtok(NULL, " \t"), NULL);
          printf("and Y=%5.1f\n", plane[i][2]);
          plane[i][3] = strtod(strtok(NULL, " \t"), NULL);
          printf("With Radius=%5.1f\n", plane[i][3]);
     }


     if(Sp4ArrayInit(&ar, nn, N_DIM, DATATYPE_COMPLEX, comments) != SUCCESS){
          printf("2ddisloc:  array init failed\n");
          return MALLOC_FAILED;
     }


     /* set all points inside the polygon to amplitude 1 and phase 0*/
     for(j=1;j<=nn[1];j++){
          for(i=1;i<=nn[2];i++){
          x=i-(int)(nn[2]/2) -1;
          y=j-(int)(nn[1]/2) -1;
          ar.data[2*(i+(j-1)*nn[2])-1]=0;
          ar.data[2*(i+(j-1)*nn[2])]=0;


               for(nt=0; nt<=N ;nt++){
               if((x - plane[nt][1] )*(x - plane[nt][1]) + (y - plane[nt][2])*(y -
plane[nt][2]) <= plane[nt][3] * plane[nt][3] )

     {
```

```
              /* sets the density of the sphere, p = sqrt(radius^2 -x^2 -y^2)/radius
*/
             R = plane[nt][3];
             x0 = (x - plane[nt][1]);
             y0 = (y - plane[nt][2]);
             /* printf(" %5.1f ",density); */

             /* additional fermi modulation, to soften edges of sphere, a0 is scaling
factor */
             a0 = 17;
             xfermi = 1/((exp(x0-a0) + 1)*(exp(-x0-a0) + 1));
             yfermi = 1/((exp(y0-a0) + 1)*(exp(-y0-a0) + 1));


            /* density = (fermi softening function) x (sphere density) */
             density = xfermi*yfermi*(sqrt(R*R - x0*x0 - y0*y0))/R;


            /* Saves real and imaginary parts */
            B = 0.96;

              ar.data[2*(i+(j-1)*nn[2])-1] = density*cos(K*y*0.01 + Phi*0.1)*(B +1);
              ar.data[2*(i+(j-1)*nn[2])]   = density*sin(K*y*0.01 + Phi*0.1)*(B -1);

          }


        }

     }



                     }


     /* name instead of argv[2] */
     if(Sp4ArraySave(&ar, argv[2]) != SUCCESS){
          printf("2ddisloc:  array save failed\n");
          return SAVE_FAILED;
     }

     Sp4ArrayDestroy(&ar);
     return SUCCESS;
}
```

```
/*  2dcirclesupport.c

     Simulate a circle, store in 2D array nn[2]ith unit amplitude.
     Used to make a circular support for phasing
     Piotr 1/1/07


Run program using command:
2dcirclesupport  <output name> <x> <y> <r>

*/

#define X_DIM 512
#define Y_DIM 512
#define N_DIM 2
#define N_PLANES 20
#define UNITY 40
#define DELTA 5
#include "sp4util.h"
#include "math.h"

int main(int argc, char* argv[]){

     int x,y,i,j,x0,y0,r0,n,it=0,a0;
     double plane[N_PLANES][3];
     double disloc[N_PLANES][4];
     double ff,density,dd,xfermi,yfermi,B;
     FILE* fp_in;
     Sp4Array ar;
     unsigned long nn[4];
     char* comments = "IKR 2/29/04 make pictures of complex dislocation strain
fields";
     char str[25],name[25];             /* each input line must be less than 25 chars
long! */
     char* string1[20];
     nn[1] = nn[2] = 896;

     /* check for usage and read in parameters */
     if(argc!=5){
          printf("Usage:\n\t2dcirclesupport <output_file> <X_position> <Y_postion>
<Radius>\n");
          return BAD_PARAMETERS;
     }

     sscanf(argv[2],"%ld",&x0);
     printf("Centre of Circle: X is %ld\n", x0);

     sscanf(argv[3],"%ld",&y0);
     printf("and Y= %ld\n", y0);

     sscanf(argv[4],"%ld",&r0);
     printf("and Radius= %ld\n", r0);

     sscanf(argv[1],"%s", &string1);

     /* was originally trying to rename output files by merging strings
     sprintf(name,"%s%d",string1,B);
          printf("The file name is %s\n",name); */



     if(Sp4ArrayInit(&ar, nn, N_DIM, DATATYPE_COMPLEX, comments) != SUCCESS){
          printf("2ddisloc:  array init failed\n");
          return MALLOC_FAILED;
     }
```

```
     /* set all points inside the polygon to amplitude 1 and phase 0*/
     for(j=1;j<=nn[1];j++){
          for(i=1;i<=nn[2];i++){
          x=i-(int)(nn[2]/2) -1;
          y=j-(int)(nn[1]/2) -1;
          ar.data[2*(i+(j-1)*nn[2])-1]=0;
          ar.data[2*(i+(j-1)*nn[2])]=0;

            /* For stuff stuff inside cirlce */

               if((x - x0 )*(x - x0) + (y - y0)*(y - y0) <= r0 * r0 )

     {
             /* Saves real and imaginary parts */

              ar.data[2*(i+(j-1)*nn[2])-1] = 1;
              ar.data[2*(i+(j-1)*nn[2])]   = 0;

          }


        }

     }

     /* name instead of argv[2] */
     if(Sp4ArraySave(&ar, argv[1]) != SUCCESS){
          printf("2ddisloc:  array save failed\n");
          return SAVE_FAILED;
     }

     Sp4ArrayDestroy(&ar);
     return SUCCESS;
}
```

 # Centre.py

```
# A program to determine FFT centre via Wrap and FTT polygon images
# Load the phasing library into python.
# Even though we aren't doing any phasing, all of the array utilities are
# useful.
# Wrappedsqrtedata.sp4 is produced for use in python phasing
# FFTimage is used to view FFT image of object

import cstuff as c
import sys

# Sp4Arrays need to be declared since they are not native to python
polyarray=c.Sp4Array()

filename=sys.argv[1]


c.Sp4ArrayLoad(polyarray, filename)
# c.Sp4ArrayShift(polyarray,1,1)
c.Sp4ArrayWrap(polyarray)
c.Sp4ArrayFFT(polyarray, +1, 1)
c.Sp4ArrayWrap(polyarray)
c.Sp4ArrayExportPPM(polyarray, "Shift_1_1")
```

Embedarray.py

```
# Load arrays
import mycstuff as mc
import cstuff as c
import sys

# Sp4Arrays need to be declared since they are not native to python
background=c.Sp4Array()
data=c.Sp4Array()
data2=c.Sp4Array()

filename = sys.argv[1]

# Create a 2d complex Sp4Array that is 896x896 elements.
# Defined values of datatype are DATATYPE_REAL
#                                 DATATYPE_COMPLEX
#                                 DATATYPE_VECTOR
# Only use DATATYPE_COMPLEX unless you know what you are doing.
# Most of the functions in the cstuff library assume this to be the case
# without checking.

# Create empty array of size 896x896
c.Sp4ArrayNew(background, c.DATATYPE_COMPLEX, 896,896, 0)
# Load data file
c.Sp4ArrayLoad(data, filename)
# Square rooting to convert from intensity to amplitude
c.Sp4ArraySqrt(data)
dx = 896/2 - 122
dy = 896/2 - 144


# EmbedArray (Sp4Array* background, Sp4Array* insert, locx, locy, locz,
# int zerobackground)
c.EmbedArray(background,data,dx,dy,0,1)


# Create PPM for viewing
c.Sp4ArrayExportPPM(background, "background")

# Wrapping the array moves the object from the center to the corners of array.
# This makes the phases sensible in the Fourier Transform.  If this is not done
# the phases after FFT will have a (-1)^(i+j) modulation, where i,j are the
# array indicies.
```

```
c.Sp4ArrayWrap(background)


c.Sp4ArraySave(background, 'wrappedsqrteddata.sp4')
```

RealFFT.py

```
# A program to Wrap and FTT polygon images
# Load the phasing library into python.
# Even though we aren't doing any phasing, all of the array utilities are
# useful.
# Wrappedsqrtedata.sp4 is produced for use in python phasing
# FFTimage is used to view FFT image of object

import mycstuff as mc
import cstuff as c
import sys

# Sp4Arrays need to be declared since they are not native to python
polyarray=c.Sp4Array()
polyarray2=c.Sp4Array()
REAL=c.Sp4Array()

filename=sys.argv[1]

# Load the arrays from files
c.Sp4ArrayLoad(polyarray, filename)

# Wrapping the array moves the object from the center to the corners of
#array.
# This makes the phases sensible in the Fourier Transform.  If this is not
#done
# the phases after FFT will have a (-1)^(i+j) modulation, where i,j are
#the
# array indicies.
c.Sp4ArrayWrap(polyarray)

# Write the wrapped poly to a file.
c.Sp4ArraySave(polyarray, "wrappedpoly.sp4")
c.Sp4ArrayExportPPM(polyarray, "wrappedpoly")

# Fourier Transform the array using the FFTw.  The transform is done
# in-place
# meaning the array is overwritten with the transform.
c.Sp4ArrayFFT(polyarray, +1, 1)

# Write the Fourier Transformed array to a file.
mc.Sp4ArrayMakeRealPG (polyarray2, polyarray)
c.Sp4ArraySave(polyarray2, "wrappedsqrteddata.sp4")
c.Sp4ArrayExportPPM(polyarray2, "wrappedsqrteddata")

# Wrapp FFT image, write and create PPM for viewing
c.Sp4ArrayWrap(polyarray2)
c.Sp4ArraySave(polyarray2, "FFTimage.sp4")
c.Sp4ArrayExportPPM(polyarray2, "FFTimage")
```

Phase.py

```
#!/usr/bin/python

# ------------------------------------------------------------------------
# load the phasing library into python
import cstuff as c
import SequenceClass as sc

# declare some variables
beta1=.9
numERiter=100

# Sp4Arrays need to be declared since they are not native to python
data=c.Sp4Array()
support=c.Sp4Array()

# Load the arrays from files
c.Sp4ArrayLoad(data, 'wrappedsqrteddata.sp4')
c.Sp4ArrayLoad(support, 'asupport.sp4')

# Going to run five sequences, each with a different initial object
for i in range(1,6):
  # The SequenceData keeps the current object and a log of what's been done
  # to it.  It is declared with an indentification number and the directory
  # where results will be written.
  seqdata=sc.SequenceData(i, './')
  if i==1:
    c.InitializeSequence_COPY(seqdata, support)
  else:
    c.InitializeSequence_dsRANDOM_REAL(seqdata, support)

  # Run the Error Reduction (ER) Algorithm for 100 iteration.
  # This particular one uses only a support constraint in direct space
  # so the result can be complex.
  c.DoER_support(seqdata, data, support, numERiter)

  # After the ER do 500 iterations of support only HIO.
  c.DofHIO_support(seqdata, data, support, beta1, 500)

  # Then another 100 iterations of the same ER algorithm
  c.DoER_support(seqdata, data, support, numERiter)

  # Write the current object and the residual array and the log file.
  seqdata.Write()
  seqdata.WritePPM()


# ------------------------------------------------------------------------
```

SPEConvert.py

```
import cstuff as c
import sys
import SpeFile as sf

filename = sys.argv[1]

File=sf.SpeFile(filename)

basefilename=filename.split('.')[0]

origarr=File.GetSp4Array()

c.Sp4ArraySave(origarr, basefilename+".sp4")

c.Sp4ArrayExportPPM(origarr, basefilename)
```