Chapter 4
Sampling and the Fast Fourier Transform

Abstract This chapter takes a small detour and discusses some numerical approximations that will be necessary to theoretically compute electron microscopy images. Specifically, digital sampling (pixels and levels) and the all important fast Fourier transform are introduced. The FFT will be the principle tool to speed up later calculation. If you are familiar with these topics, this chapter may be skipped.

To go further with image calculations requires a detailed numerical calculation using a computer program. The mathematics gets too complicated (or long) to perform analytically with pencil and paper. This chapter gives some necessary computer background prior to calculating images in later chapters. The computer imposes its own set of rules that must be understood and dealt with to perform these calculation. Experienced computer user may prefer to skip this chapter.

Image simulation or image processing with the computer presumes that the image is somehow represented inside the computer. A digital computer naturally operates on numerical data. Therefore, an image must be represented as a two dimensional array of numbers inside the computer. Each number is one pixel or spot in the image whose intensity is proportional to its numerical value. The trick is to have a sufficiently large number of pixels so that when they are displayed as an image the individual numbers or pixels are not individually distinguishable. Sampling the image in this manner leads to some specific rules and limitation that are summarized in this chapter.

The fast Fourier transform or FFT is one of the most efficient computer algorithms available. The FFT computes the Fourier transform of discretely sampled data in a minimum amount of computer time. Image simulation, such as the multislice method, are usually organized around the FFT to reduce the computer time required for simulations. The mechanism of the FFT is closely coupled to discretely sampled data and is also included in this chapter.

E.J. Kirkland, *Advanced Computing in Electron Microscopy*,
DOI 10.1007/978-1-4419-6533-2_4,

4.1 Sampling

In practice each image is calculated in a rectangular grid of $N_x \times N_y$ pixels or picture elements as shown in Fig. 4.1. The images are sampled at N_x discrete points along x and N_y discrete points along y and form a supercell with dimensions of $a \times b$ in real space. Figure 4.2 shows the visual effects of changing the number of pixels in an image for the special case of $N_x = N_y$ and $a = b$. Each pixel has dimensions $a/N_x \times b/N_y$ in real space and has a single value associated with it that is the average over the area of the pixel (or in the case of a complex wave function two values representing the real and imaginary parts). The real space coordinates take on only discrete values of:

$$x = i\Delta x \quad i = 0, 1, 2, \cdots, (N_x - 1) \tag{4.1}$$

$$y = j\Delta y \quad j = 0, 1, 2, \cdots, (N_y - 1), \tag{4.2}$$

where $\Delta x = a/N_x$ and $\Delta y = b/N_y$.

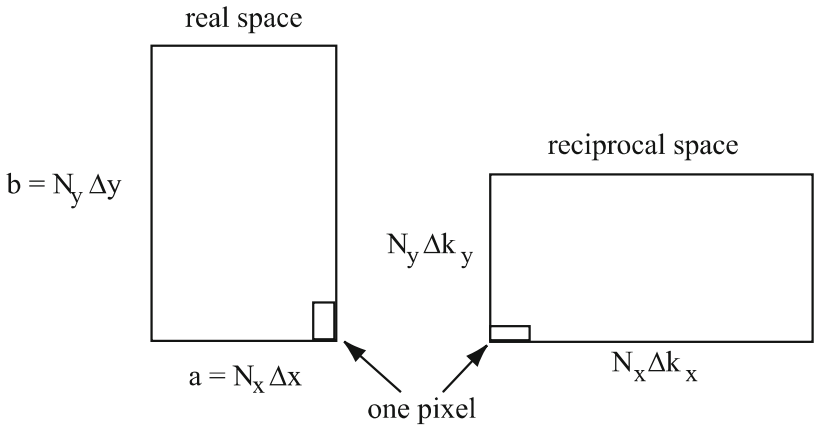

Fig. 4.1 Sampling of an image of size $a \times b$ in a plane perpendicular to the optic axis of the microscope. There are $N_x \times N_y$ pixels with a size of $a/N_x \times b/N_y$ in real space and $1/a \times 1/b$ in reciprocal or Fourier transform space. Neither the pixels or the image have to be *square*

The Fourier transform of this image will also have $N_x \times N_y$ pixels but the dimensions of each pixels change to $1/a \times 1/b$ and the reciprocal space coordinates take on values:

$$k_x = i\Delta k_x \quad i = 0, 1, 2, \cdots, (N_x - 1) \tag{4.3}$$

$$k_y = j\Delta k_y \quad j = 0, 1, 2, \cdots, (N_y - 1), \tag{4.4}$$

where $\Delta k_x = 1/a$ and $\Delta k_y = 1/b$. The supercell does not have to be square and there may be a different number of pixels in x and y although this is usually less efficient. It is interesting to note that the longest dimension of the supercell will be reversed in real and reciprocal space (i.e., a tall narrow supercell in real space becomes short and wide in reciprocal space).

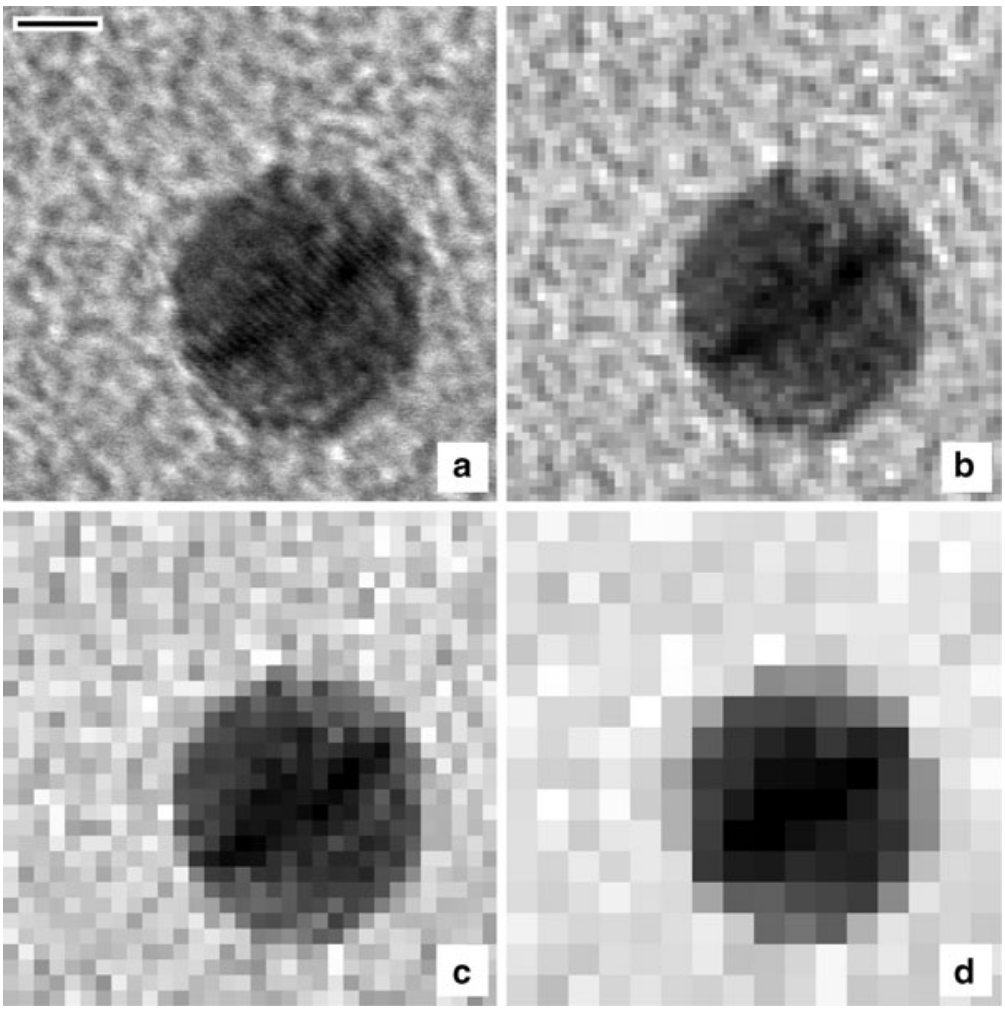

Fig. 4.2 The visual effects of changing the number of pixel in an image. (a) 256×256 pixels, (b) 64×64 pixels, (c) 32×32 pixels, and (d) 16×16 pixels. Each pixel has eight bits. The image is a bright field STEM image of a *gold particle* on an amorphous carbon film recorded on a VG HB-501 STEM ($C_s = 1.3$ mm, 100 keV). The 2.35Å gold lattice fringes are barely visible in the center of the particle. The scale bar in (a) is approximately 20Å

Discrete sampling also imposes a limit on the maximum spatial frequency in the image of:

$$|k_x| < \frac{1}{2\Delta x}$$
$$|k_y| < \frac{1}{2\Delta y} \tag{4.5}$$

This is referred to as the Nyquist limit. In principle this limit may be different in each direction, however in practice the larger limit should be reduced to the smaller so that the effective resolution is isotropic in the image (i.e., to avoid sampling artifacts in the image). If the sampling size Δx or Δy is too large then the signal is under sampled and aliasing occurs. Figure 4.3 shows the effect of under sampling a sine wave. The high frequency sine wave appears to be a low frequency sine wave if it is under sampled. This should be avoided by making the sampling size smaller or explicitly limiting the bandwidth of the original signal.

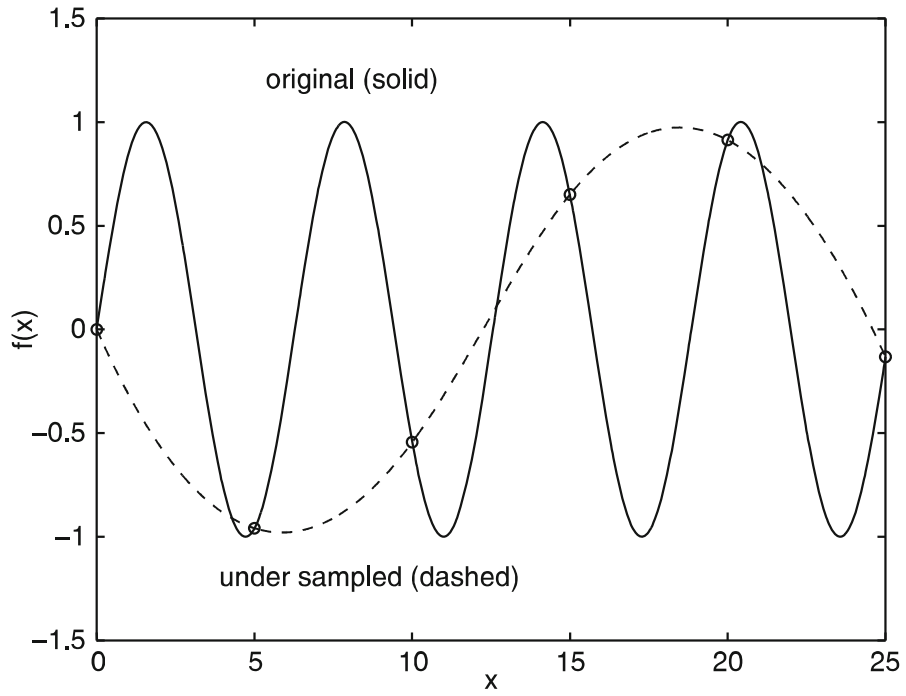

Fig. 4.3 The affect of under sampling a function. The original function is shown as a *solid line*. It is under sampled at regular intervals (*open circles*). The apparent signal is shown as a *dashed line*. Under sampling causes high frequencies to be improperly aliased as low frequencies

The numerical value associated with each pixel is itself discretely sampled. This value must be encoded as a finite number of bits (one bit has a value of 0 or 1) for each pixel in the computer. The human eye can distinguish at best about 30–50 different shades of grey. Figure 4.4 shows the effects on human perception of varying the number of bits in each pixel. The computer hardware is usually capable of conveniently handling data in units as small as 8 bits at a time. Therefore, about 8 bits of information is needed for each pixel to display the image. Any substantial amount of calculation with only 8 bits per pixels will however quickly get into a lot of trouble. Rounding each value to 8 bits introduces an error of at least one in $2^8 = 256$. Even worse, dividing two eight bit integer numbers truncates the result to the lowest integer (for example $1/2 = 0$, $128/50 = 2$, with integer arithmetic) which can introduce very large errors for each arithmetic operation between pixels. An image simulation may require thousands or millions of operations on each pixel. Therefore, during image simulation each pixel should be represented as a floating point number with much more than 8 bits per pixel. Most computer hardware is equipped to handle 32 bit single precision (typically there is one sign bit, 8 exponent bits and 23 mantissa bits) and 64 bit double precision floating point arithmetic. Single precision (32 bits) gives about six decimal digits of accuracy per pixel and is usually sufficient for most image calculations. Each arithmetic operation between two single precision floating point numbers can be thought of as adding an error of about ± 1 in 10^6 or 1 in 6 digits. This error is sometimes referred to as round-off error, and calculations with a finite number of bits is sometimes referred to as finite precision arithmetic. Although this error may seem insignificant it may be necessary to

perform a million operations on some numbers so the errors can add up to be significant even with single precision (32 bit) floating point arithmetic. A good computer program should be organized in such a way as to minimize the effects of round-off error. Storing several images of sizes of 512×512 pixels or more requires a lot of memory so double precision is usually not used to keep the memory requirements to a reasonable level. During numerical simulation each pixels should be stored as a 32 bit (or more) floating point number and for the final displayed result 8 bits is probably sufficient.

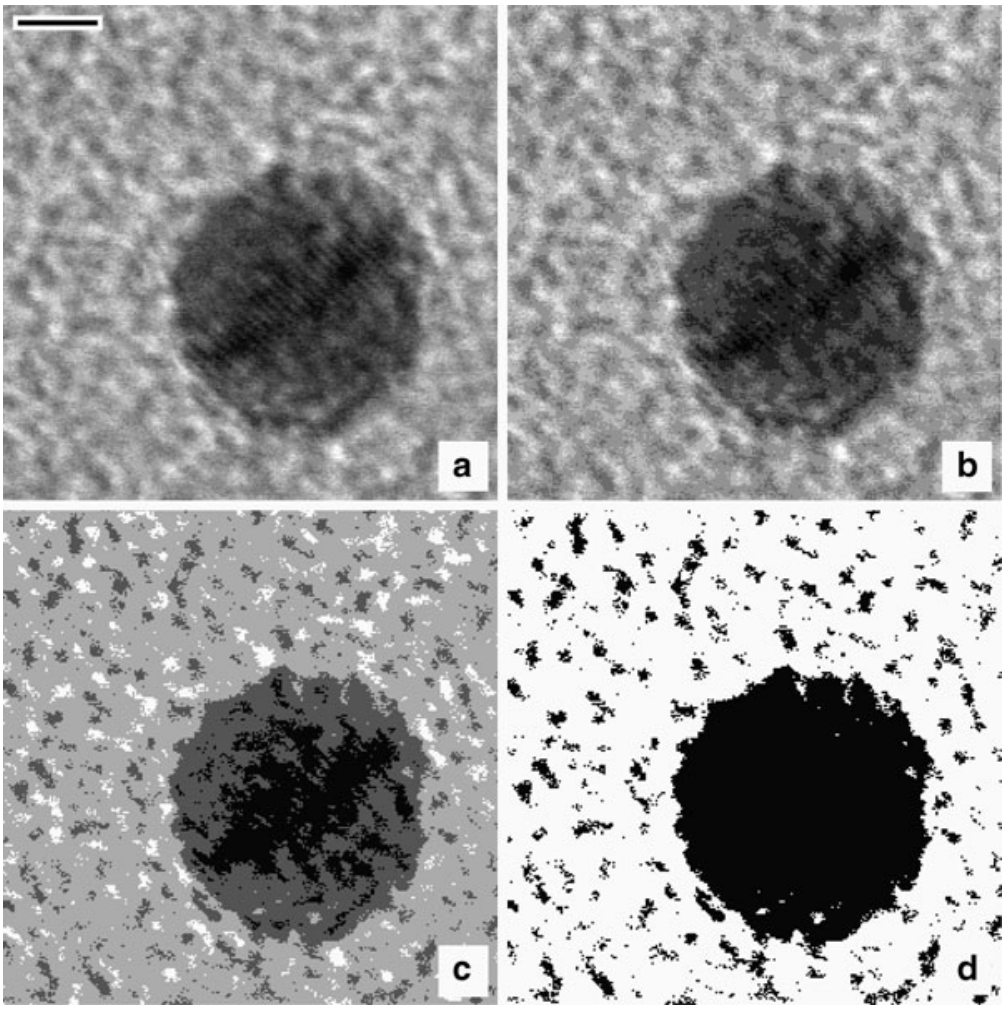

Fig. 4.4 The effect of limiting the number of bits in each pixel. (a) 4 bits/pixel, (b) 3 bits/pixel, (c) 2 bits/pixel, and (d) 1 bits/pixel. Each image was normalized to fill the available greyscale. The image is a bright field STEM image of a gold particle on an amorphous carbon film recorded on a VG HB-501 STEM ($C_s = 1.3$ mm, 100 keV). The 2.35Å gold lattice fringes are barely visible in the center of the particle. The scale bar in (a) is approximately 20Å. (Fig. 4.2 a is the same image with 8 bits per pixel)

4.2 Discrete Fourier Transform

Image simulation and image processing frequently uses a discrete Fourier transform (DFT) to perform convolutions or to convert from reciprocal space to real space and vice versa. There are several different ways to define a DFT that differ mainly in the placement of the minus signs and normalization constants. The specific definition of the Fourier transform FT and its inverse FT^{-1} that will be used here is:

$$\mathrm{FT}[f(\mathbf{x})] = F(\mathbf{k}) = \sum_{x,y} \exp(2\pi \mathrm{i}\mathbf{k}\cdot\mathbf{x}) f(\mathbf{x}) \tag{4.6}$$

$$\mathrm{FT}^{-1}[F(\mathbf{k})] = f(\mathbf{x}) = \frac{1}{N_x N_y} \sum_{k_x,k_y} \exp(-2\pi \mathrm{i}\mathbf{k}\cdot\mathbf{x}) F(\mathbf{k}) \tag{4.7}$$

where $\mathbf{x} = (x,y)$ and $\mathbf{k} = (k_x,k_y)$. The inverse Fourier transform can be written as the complex conjugate of the forward transform of the complex conjugate of $F(\mathbf{k})$.

$$\mathrm{FT}^{-1}[F(\mathbf{k})] = \frac{1}{N_x N_y} \left\{\mathrm{FT}[F^*(\mathbf{k})]\right\}^* . \tag{4.8}$$

It is only necessary to program one or the other transform (forward or inverse) and the other can be obtained with suitable complex conjugation and scaling.

When expressed in (x,y) Cartesian coordinates the Fourier transform is separable in x and y.

$$F(k_x,k_y) = \sum_x \exp(2\pi \mathrm{i} k_x x) \left[\sum_y \exp(2\pi \mathrm{i} k_y y) f(x,y)\right] \tag{4.9}$$

A two dimensional transform (forward or inverse) may be implemented by successive one-dimensional transforms. First perform a one dimensional transform along all of the columns and then along all of the rows (or vice versa) of the sampled image (row-column decomposition). Therefore it is only necessary to program a single one-dimensional transform to perform both forward and inverse transforms on two dimensional images. It is much easier in practice to arrange the images to be sampled in a rectangular (x,y) Cartesian grid to exploit the separability of the Fourier transform. Also note that the spacing in x and y may be different but it is usually advisable that the x and y spacings be within a factor of two or so of each other.

4.3 The Fast Fourier Transform or FFT

The one-dimensional discrete Fourier transform is:

$$F(n\Delta k) = F_n = \sum_j f(j\Delta x) \exp[2\pi \mathrm{i}(n\Delta k)(j\Delta x)] \tag{4.10}$$

$$k = n\Delta k; \quad \Delta k = 1/a; \quad n = 0,1,2,\cdots,(N-1)$$

$$x = j\Delta x; \quad \Delta x = a/N; \quad j = 0,1,2,\cdots,(N-1)$$

with $f_j = f(j\Delta x)$ this simplifies to:

$$F_n = \sum_j f_j \exp[2\pi \mathrm{i}(nj/N)] \tag{4.11}$$

The amount of computer time this requires typically scales as the number of floating point operations such as add, subtract, multiply, and divide. A one-dimensional DFT of length N requires N sums each with N terms and thus requires a computer time that is proportional to N^2.

The computer time may be greatly reduced by use of the fast Fourier transform or FFT algorithm (Cooley and Tukey [53], Brigham [39], and Bracewell [37]). The FFT requires that the length N of the transform be a highly composite number (i.e., be factorable into many smaller integer prime factors). Usually factors of two are a little more efficient although this is not a strict requirement. If the length of the data array is some power of two, $N = 2^m$ (N and m integer) then the data array index j may be written as:

$$j = j_0 + j_1 2 + j_2 2^2 + j_3 2^3 + \cdots + j_{m-1} 2^{m-1}, \tag{4.12}$$

where each of the $j_0, j_1, \ldots$ takes on values of 0 or 1. Equation (4.11) can then be written as:

$$\begin{aligned} F_n \; = & \sum_{j_0} \sum_{j_1} \sum_{j_2} \cdots \sum_{j_{m-1}} f_{j_0 j_1 j_2 \cdots j_{m-1}} \\ & \exp[2\pi \mathrm{i} n(j_0 + j_1 2 + j_2 2^2 + \cdots + j_{m-1} 2^{m-1})/N]. \end{aligned} \tag{4.13}$$

The sums can be rearranged as:

$$\begin{aligned} F_n \; = & \sum_{j_0} \exp[2\pi \mathrm{i} n j_0/N] \sum_{j_1} \exp[2\pi \mathrm{i} n j_1 2/N] \\ & \cdots \sum_{j_{m-1}} \exp[2\pi \mathrm{i} n j_{m-1} 2^{m-1}/N] f_{j_0 j_1 j_2 \cdots j_{m-1}}. \end{aligned} \tag{4.14}$$

At first glance it seems as if this has just gotten a lot more complicated without any gain. However a close inspection of the arithmetic reveals that there are $m = \log_2 N$ sums each with two terms. Each of the N Fourier components F_n requires m sums of length 2 so the total computer time becomes proportional to $Nm = N \log_2 N$. This is a huge savings in computer time for large N.

When N is decomposed into factors of 2 the FFT is said to be a radix-2 FFT. The radix-2 FFT requires frequent multiplication by sine and cosine of 0 and π (equals 0 and ± 1) which can be hand coded to avoid some floating point arithmetic operations. Curiously, if factors of 4 are also treated as prime factors, sine and cosine of 0, $\pi/2$, $3\pi/2$ and π appear (equals 0, ± 1 and $\pm i$). These can be hand coded without floating point arithmetic to produce an FFT that is a little faster than using only factors of 2. Factors of 8 also give a very small improvement in speed but significantly increase the code size so only factors of 2 and 4 are commonly treated. When factors of

four are used the FFT is said to be a radix-4 FFT. When N is decomposed into both factors of 2 and 4 the FFT is said to be a mixed radix FFT. It would first do all of the factors of 4 and then at most one factor of 2 to get any length that is a power of 2. If a two dimensional (or higher) FFT is needed, an additional improvement in performance can also be obtained by using a look up table for the sines and cosines because they only need to be calculated once. Higher radix FFT's generally trade integer operations for floating point operation. Until recently floating point operations were almost always much slower than integer operation and higher radix FFT's run faster. Some new computer architectures close the gap (in speed) between integer and floating and it is possible that radix-4 or radix-8 may not be significantly faster than radix-2 on some specific types of computers if floating point and integer operations are equally fast. Multidimensional transforms can also be limited by the memory bandwidth, and may benefit from careful attension to the order in which the data is accessed. The FFT is still a DFT in some sense (the FFT also satisfies (4.6) but just does it faster) but the two names will be used to distinguish the simple sum from the fast form of the sum.

The computer time for a radix-2 two dimensional transform of length $N_x \times N_y$ for the DFT and the FFT (4.6) is approximately proportional to:

$$\text{CPU time for simple DFT} \propto N_y N_x^2 + N_x N_y^2 \tag{4.15}$$

$$\text{CPU time for FFT} \propto N_x N_y \log_2(N_x N_y). \tag{4.16}$$

The constant of proportionally is of order unity in both cases. The advantage of the FFT over the DFT is very large as N gets bigger. Table 4.1 illustrates the relative CPU time of the DFT vs. the FFT for some typical lengths of the data array. In two dimensions the ratio of the FFT to the DFT remains the same as in the table if $N_x = N_y = N$.

Table 4.1 Comparison of the relative CPU time required for a simple discrete Fourier transform (DFT) and a fast Fourier transform (FFT) for different lengths N

Number of data points N	$\log_2(N)$	DFT	FFT	Ratio
32	5	1024	160	6.4
64	6	4096	384	10.7
128	7	16,384	896	18.3
256	8	65,536	2048	32
512	9	262,144	4608	56.9
1024	10	1,048,576	10240	102.4

The FFT is the workhorse of image simulation and image processing. It is worthwhile to optimize the code for efficient execution in the computer because it gets used over and over again. There are a variety of other tricks that can be incorporated into an actual program. The book by Brigham [39] gives an excellent discussion of strategies for implementing an efficient FFT. Working code for the FFT has been given by Press et al. [288] and Gonzalez and Wintz [124, 125]. A version of a one dimensional FFT using a mixed radix-2 and radix-4 approach in C is given at the

end of this chapter. There are many freely available FFT subroutines available to download. The currently popular FFTW package (www.fftw.org [115]) seems to be one of the fastest (if not the fastest) available code. Multidimensional FFTs can be implemented by successively applying a 1D FFT to each dimension. In a 2D FFT each row and then each column. This procedure is easy to adapt to a multiply CPU (multithreaded) computer. Each row (or subset of rows) is independent so can be done simultaneously on a different CPU (many rows at the same time) and then each column (or subset of columns) on a different CPU. The openMP syntax for multithreading is relatively easy to use and has become commonly implemented in several different compilers and computer (and is mostly platform independent).

4.4 Wrap Around Error and Rearrangement

A consequence of a discrete Fourier transform (DFT or FFT) is that the sampled data is repeated indefinitely in a periodic array. An identical copy of the image appears on all four sides of the image. The image is usually not drawn this way but is implied by the use of discrete sampling and the discrete Fourier transform. In practice this periodic repetition means that the left and right edges of the image are effectively adjacent to one another and can interfere ($x = 0$ is equivalent to $x = \mathrm{a}$, $x = 2\mathrm{a}$, etc.). The same is true of the top and bottom of the image. This effect is called the wrap-around error because the left and right or top and bottom edges effectively wrap around and touch each other. If the underlying specimen does not share this periodicity then some rather dramatic artifacts that have nothing to do with the specimen can be produced in simulated images. The underlying specimen periodicity should match the periodicity of the supercell (the supercell dimensions should be an integer multiple of the specimen unit cell size if the specimen is crystalline) or there should be a buffer zone around the edge of the image that can be discarded later. The width of this buffer zone varies with the application and may not always be obvious.

The wrap around effect applies in both real space and reciprocal space. This causes a strange distribution of spatial frequencies in the FFT. Large positive frequencies are the same as small negative frequencies. The sampled frequencies in reciprocal space (4.4) may be written in order from left to right (in x) and bottom to top (in y) as:

$$k_x = 0, \Delta k_x, 2\Delta k_x, 3\Delta k_x, \cdots, (N_x - 1)\Delta k_x \tag{4.17}$$

$$k_y = 0, \Delta k_y, 2\Delta k_y, 3\Delta k_y, \cdots, (N_y - 1)\Delta k_y. \tag{4.18}$$

With wrap around the $k_x = (N_x - 1)\Delta k_x$ position is touching the $k_x = 0$ position on the left. This means that the $N_x - 1$ position is the same as the -1 position (likewise in the y direction). A circle with constant magnitude of spatial is drawn in FFT space on the left side of Fig. 4.5 (labeled default). The origin is in the lower left corner.

What is normally the first quadrant is in the lower left corner, the second quadrant is in the lower right, the third quadrant in the upper right and the fourth quadrant in the upper left. The distribution of spatial frequencies is normally used in this order during calculation because it would waste computer time to rearrange it. However when displayed the FFT will be rearranged (as on the right hand side of Fig. 4.5) to conform to a normal diffraction pattern with zero spatial frequency in the center.

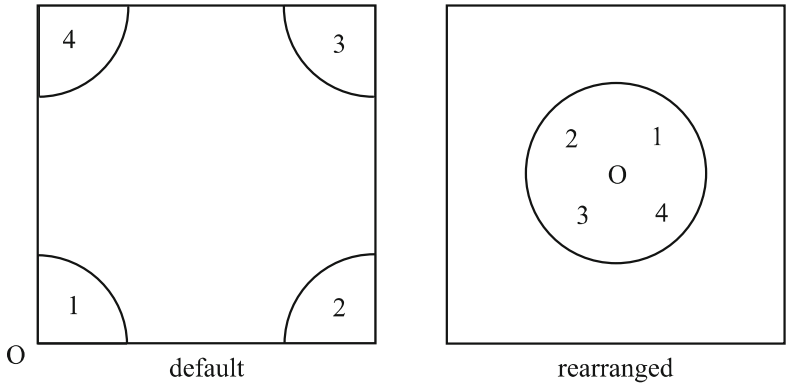

Fig. 4.5 Geometry of the FFT (or DFT). A *circle* of constant spatial frequency is drawn in the default configuration (*left*) and the rearranged configuration (*right*). The origin is at the lower left in the default configuration and in the center in the configuration at right rearranged for display purposes

4.5 Fourier Transforming Real Valued Data

The FFT discussed in Sect. 4.3 assumed a general case in which the data $f(x)$ is complex valued with a real and imaginary part (as in an electron wave function). Sometimes the data will be explicitly real valued (as in the potential $V(\mathbf{r})$ of the specimen). The special case of Fourier transforming real data allows a further reduction in computer time of about a factor of two. There are two possible methods to use. One is to transform a single real valued array more efficiently and the other is to transform more than one real array at a time with a single complex FFT. Both of these methods are discussed in Press et al. [288]. Performing a two dimensional FFT on real valued data is easiest using the second of these methods.

Consider two one dimensional arrays of N real values, $f_a(x)$ and $f_b(x)$, linearly combined into a single complex valued array $f_c(x)$ as:

$$f_c(x) = f_a(x) + i f_b(x) \tag{4.19}$$

$f_c(x)$ is now in the form that can be used in a single FFT to produce N complex valued Fourier coefficients. The trick is to untangle the Fourier transform of $f_a(x)$ from that of $f_b(x)$. In real space $f_a(x)$ and $f_b(x)$ can be retrieved from $f_c(x)$ using (4.19) as:

$$f_a(x) = [f_c(x) + f_c^*(x)]/2 \tag{4.20}$$

$$f_b(x) = [f_c(x) - f_c^*(x)]/(2i) \tag{4.21}$$

In Fourier or reciprocal space:

$$F_c(k) = FT[f_c(x)] = \int f_c(x) \exp(2\pi \mathrm{i} kx) \mathrm{d}x \tag{4.22}$$

$$F_c^*(-k) = FT[f_c^*(x)] = \int f_c^*(x) \exp(2\pi \mathrm{i} kx) \mathrm{d}x \tag{4.23}$$

Therefore, the transforms of $f_a(x)$ and $f_b(x)$ may be extracted from the transform of $f_c(x)$ [using (4.21)] as:

$$FT[f_a(x)] = F_a(k) = [F_c(k) + F_c^*(-k)]/2 \tag{4.24}$$

$$FT[f_b(x)] = F_b(k) = [F_c(k) - F_c^*(-k)]/(2i) \tag{4.25}$$

If there are N real values in each of $f_a(x)$ and $f_b(x)$ then there are N complex values in $f_c(x)$ and $F_c(k)$. It follows that there are $N/2$ complex valued Fourier coefficients in each of $F_a(k)$ and $F_b(k)$ because:

$$F_a(k) = F_a^*(-k) \tag{4.26}$$

$$F_b(k) = F_b^*(-k) \tag{4.27}$$

To calculate a two dimensional Fourier transform of $N_x \times N_y$ real valued data points, first calculate the Fourier transform of all of the N_y columns two at a time, yielding N_y complex arrays of length $N_x/2$. Next Fourier transform N_x rows of $N_x/2$ complex values. This is referred to as a real to complex FFT and results in a net sped up of about a factor of two.

4.6 Displaying Diffraction Patterns

The square modulus of the Fourier transform of a function is called its power spectra. The power spectra of the wave function transmitted through the specimen is also equivalent to the electron diffraction pattern of the specimen. A diffraction pattern typically has a very large dynamic range in its intensity. The low spatial frequency information (low scattering angle) has a large amplitude but the high spatial frequency information (high scattering angle) has a much lower amplitude. A normal image display device (computer screen or printed paper) does not have a sufficient dynamic range to display both sets of information. The high spatial frequency information (which is frequently the interesting part) is not visible if the diffraction pattern is normalized to fill the available grey scale in a linear manner. In practice when a diffraction pattern is recorded the film or other detector may be

saturated near the central beam to produce a nonlinear scale or multiple patterns may be recorded at different exposures to accommodate this large dynamic range. Gonzalez and Wintz [124, 125] and Pratt [287] suggest that a numerically calculated power spectra be displayed on a logarithmic scale to compress the dynamic range so that the entire diffraction pattern is visible. A simple logarithm will not work because some points of the power spectra (diffraction pattern) may be identically zero. The computer program must somehow limit the negative extent of the image scale. One method is to simply clip all negative values (of the logarithm) to some minimum value. For example, the minimum grey scale can be set to the average value in some region of reciprocal space about half way between the minimum and maximum spatial frequencies. An alternative (similar to that proposed by Gonzalez and Wintz [124, 125] and Pratt [287]) is to transform the intensities as:

$$D(k_x, k_y) = \log(1 + c|F(k_x, k_y)|^2), \tag{4.28}$$

where $F(k_x, k_y)$ is the Fourier transform of the image, $D(k_x, k_y)$ is the actual value displayed and c is a scaling constant that can be varied to adjust the contrast. Figure 4.6 shows the power spectra of the electron wave function transmitted through approximately 100Å of silicon in the 110 orientation at an electron energy of 100 keV (simulated using the multislice method discussed in later chapters and a super cell size of 27×27Å with 128×128 pixels). The linear grey scale in Fig. 4.6a shows only the zero order beam in the center (rearranged as in Sect. 4.4) plus a few of the low order reflections. When displayed using (4.28) with c = 0.1 the higher order diffraction spots become visible in Fig. 4.6b. Most diffraction patterns shown in this book will use a compressed scale something like this.

4.7 An FFT Subroutine in C

The FFT is one of the most efficient (fast computation) algorithms available and is one of the primary drivers of the multislice algorithm to be considered in later chapters. There are many FFT subroutines available for downloading or purchase alone or as part of standard libraries. Below is the code for a simple one dimensional FFT. Multidimensional transforms are typically performed as successive 1D FFT's in each direction, and can be easily implemented in parallel in a shared memory multiprocessor computer

```
/*----------------------- fft42 -------------------------

    fft42( fr[], fi[], n )        radix-4,2 FFT in C

    fr[], fi[] = (float) real and imag. array with input data
    n      = (long) size of array

  calculate the complex fast Fourier transform of (fr,fi)
  input array fr,fi (real,imaginary) indexed from 0 to (n-1)
  on output fr,fi contains the transform
```

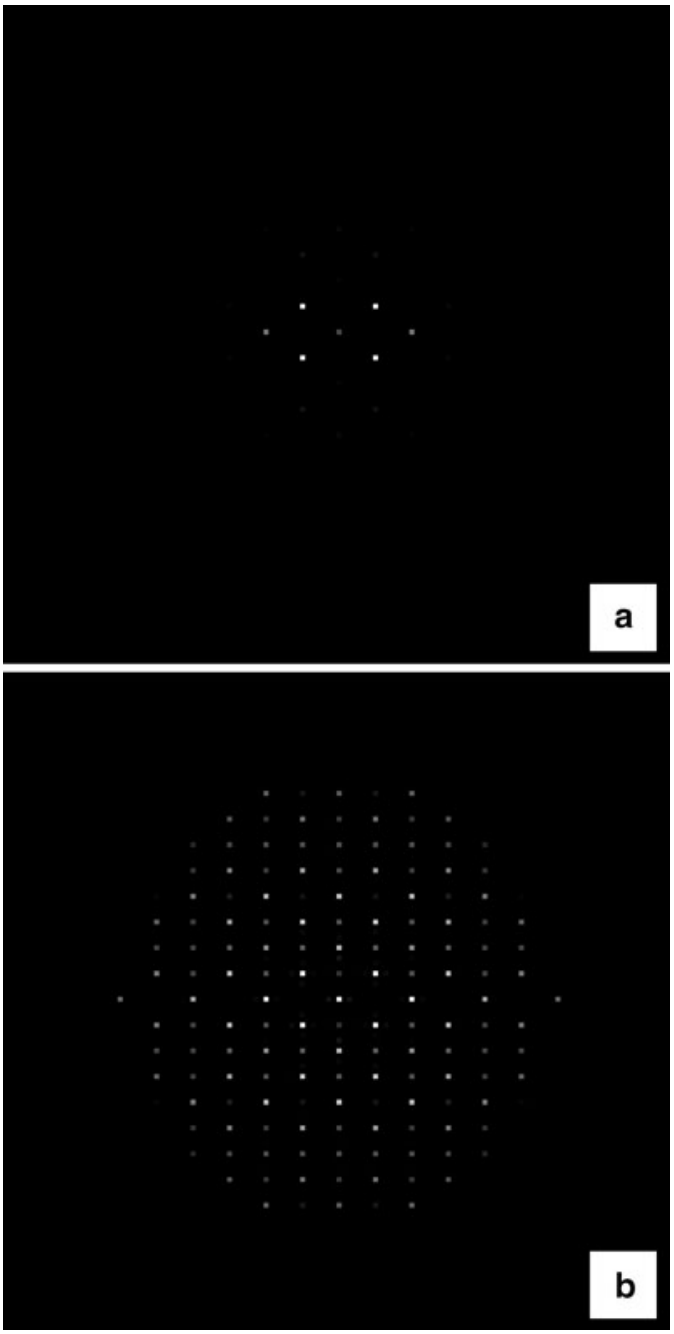

Fig. 4.6 The calculated diffraction pattern or power spectra of 110 silicon. (a) Is shown on a linear scale and (b) is a logarithmic scale as in (4.28) White is a larger positive value

```
*/

void fft42 ( float *fr, float *fi, long n  )
{
#define TWOPI   6.283185307

    long i, j, nv2, nm1, k, k0, k1, k2, k3, kinc, kinc2;
    float qr, qi, rr, ri, sr, si, tr, ti, ur, ui;
    double x1, w0r, w0i, w1r, w1i, w2r, w2i, w3r, w3i;
```

```
kinc = n;

while( kinc >= 4 ) {    /* start radix-4 section */

    kinc2 = kinc;
    kinc = kinc / 4;

    for( k0=0; k0<n; k0+=kinc2) {
        k1 = k0 + kinc;
        k2 = k1 + kinc;
        k3 = k2 + kinc;

        rr =  fr[k0] + fr[k2];    ri = fi[k0] + fi[k2];
        sr =  fr[k0] - fr[k2];    si = fi[k0] - fi[k2];
        tr =  fr[k1] + fr[k3];    ti = fi[k1] + fi[k3];
        ur = -fi[k1] + fi[k3];    ui = fr[k1] - fr[k3];

        fr[k0] = rr + tr;    fi[k0] = ri + ti;
        fr[k2] = sr + ur;    fi[k2] = si + ui;
        fr[k1] = rr - tr;    fi[k1] = ri - ti;
        fr[k3] = sr - ur;    fi[k3] = si - ui;
    }

    x1 = TWOPI/( (double) kinc2 );
    w0r = cos( x1 );   w0i = sin( x1 );
    w1r = 1.0;   w1i = 0.0;

    for( i=1; i<kinc; i++) {
        x1 = w0r*w1r - w0i*w1i;    w1i = w0r*w1i + w0i*w1r;
        w1r = x1;
        w2r = w1r*w1r - w1i*w1i;   w2i = w1r*w1i + w1i*w1r;
        w3r = w2r*w1r - w2i*w1i;   w3i = w2r*w1i + w2i*w1r;

        for( k0=i; k0<n; k0+=kinc2) {
            k1 = k0 + kinc;
            k2 = k1 + kinc;
            k3 = k2 + kinc;

            rr =  fr[k0] + fr[k2];    ri = fi[k0] + fi[k2];
            sr =  fr[k0] - fr[k2];    si = fi[k0] - fi[k2];
            tr =  fr[k1] + fr[k3];    ti = fi[k1] + fi[k3];
            ur = -fi[k1] + fi[k3];    ui = fr[k1] - fr[k3];

            fr[k0] = rr + tr;    fi[k0] = ri + ti;

            qr = sr + ur;    qi = si + ui;
            fr[k2] = (float) (qr*w1r - qi*w1i);
            fi[k2] = (float) (qr*w1i + qi*w1r);

            qr = rr - tr;    qi = ri - ti;
            fr[k1] = (float) (qr*w2r - qi*w2i);
            fi[k1] = (float) (qr*w2i + qi*w2r);

            qr = sr - ur;    qi = si - ui;
```

```
                fr[k3] = (float) (qr*w3r - qi*w3i);
                fi[k3] = (float) (qr*w3i + qi*w3r);
            }
        }

    }  /*  end radix-4 section */

    while( kinc >= 2 ) {    /* start radix-2 section */

        kinc2 = kinc;
        kinc = kinc /2 ;

        x1 = TWOPI/( (double) kinc2 );
        w0r = cos( x1 );   w0i = sin( x1 );
        w1r = 1.0;   w1i = 0.0;

        for( k0=0; k0<n; k0+=kinc2 ){
            k1 = k0 + kinc;
            tr = fr[k0] - fr[k1];  ti = fi[k0] - fi[k1];
            fr[k0] = fr[k0] + fr[k1];
            fi[k0] = fi[k0] + fi[k1];
            fr[k1] = tr;  fi[k1] = ti;
        }

        for( i=1; i<kinc; i++) {
            x1 = w0r*w1r - w0i*w1i;  w1i = w0r*w1i + w0i*w1r;
            w1r = x1;
            for( k0=i; k0<n; k0+=kinc2 ){
                k1 = k0 + kinc;
                tr = fr[k0] - fr[k1];    ti = fi[k0] - fi[k1];
                fr[k0] = fr[k0] + fr[k1];
                fi[k0] = fi[k0] + fi[k1];
                fr[k1] = (float) (tr*w1r - ti*w1i);
                fi[k1] = (float) (tr*w1i + ti*w1r);
            }
        }

    }  /* end radix-2 section */

    nv2 = n / 2;
    nm1 = n - 1;
    j = 0;

    for (i=0; i< nm1; i++) {  /* reorder in bit rev. order */
        if( i < j ){
            tr = fr[j]; ti = fi[j];
            fr[j] = fr[i];  fi[j] = fi[i];
            fr[i] = tr; fi[i] = ti; }
        k = nv2;
        while ( k <= j ) { j -=  k;  k = k>>1; }
        /* while ( k <= j ) {j=j-k;  k= k /2; }  is slower */
        j += k;
    }
```

```
#undef TWOPI
}  /* end fft42() */
```

4.8 Further Reading

Some Books on Computer Image Processing

1. K. R. Castleman, *Digital Image Processing*, Prentice Hall, 1979 [46]
2. R.C. Gonzalez and R.E. Woods, *Digital Image Processing, 3nd edition*, Prentice-Hall, 2008 [125]
3. E.L. Hall, *Computer Image Processing and Recognition*, Academic Press, 1979 [143]
4. B. Jahne, *Digital Image Processing, 3rd edition*, Springer, 1995 [183]
5. A. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, 1989 [184]
6. D.L. Missel, *Image Analysis, Enhancement annd Interpretation*, North Holland, 1978 [243]
7. W.K. Pratt, *Digital Image Processing*, Wiley, 1978 [287]
8. A. Rosenfeld and A.C. Kak,*Digital Picture Processing*, Academic Press, 1976 [305]
9. W.O. Saxton, *Computer Techniques for Image Processing in Electron Microscopy, Adv. in Electronics and Electron Physics, Supplement 10*, Academic Press, 1978 [309]

Some Books on Fourier Transforms and Fourier Optics

1. E.O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, 1974 [39]
2. J.W. Goodman, *Intro. to Fourier Optics, 3rd. edit.*, Roberts and Co., 2005 [126]
3. James S. Walker, *Fast Fourier Transforms, 2nd edit.*, CRC Press, 1996 [362]