The Fourier Transform

The two-dimensional DFT

In two dimensions, the DFT takes a matrix as input, and returns another matrix, of the same size, as output. If the original matrix values are $f(x, y)$, where x and y are the indices, then the output matrix values are $F(u, v)$. We call the matrix F the *Fourier transform of* f and write

$$F = \mathcal{F}(f).$$

Then the original matrix f is the *inverse Fourier transform of* F, and we write

$$f = \mathcal{F}^{-1}(F).$$

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\exp\left[-2\pi i\left(\frac{xu}{M}+\frac{yv}{N}\right)\right]. \tag{4.4}$$

$$f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)\exp\left[2\pi i\left(\frac{xu}{M}+\frac{yv}{N}\right)\right]. \tag{4.5}$$

Some properties of the two dimensional Fourier transform

Separability. Notice that the Fourier transform "filter elements" can be expressed as products:

$$\exp\left[2\pi i\left(\frac{xu}{M}+\frac{yv}{N}\right)\right]=\exp\left[2\pi i\frac{xu}{M}\right]\exp\left[2\pi i\frac{yv}{N}\right].$$

The first product value

$$\exp\left[2\pi i\frac{xu}{M}\right]$$

depends only on x and u, and is independent of y and v. Conversely, the second product value

$$\exp\left[2\pi i\frac{yv}{N}\right]$$

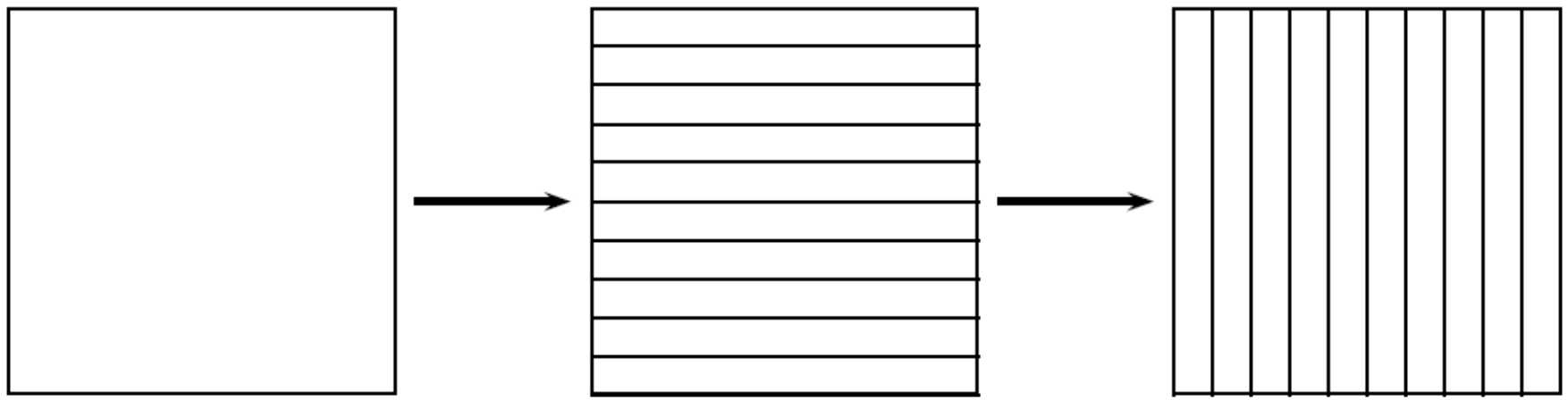
depends only on y and v, and is independent of x and u. This means that we can break down our formulas above to simpler formulas that work on single rows or columns:

$$F(u) = \sum_{x=0}^{M-1} f(x)\exp\left[-2\pi i\frac{xu}{M}\right], \tag{4.6}$$

$$f(x) = \frac{1}{M}\sum_{u=0}^{M-1} F(u)\exp\left[2\pi i\frac{xu}{M}\right]. \tag{4.7}$$

If we replace x and u with y and v we obtain the corresponding formulas for the DFT of matrix columns. These formulas define the *one-dimensional DFT* of a vector, or simply the DFT.

The 2-D DFT can be calculated by using this property of "separability"; to obtain the 2-D DFT of a matrix, we first calculate the DFT of all the rows, and then calculate the DFT of all the columns of the result, as shown in figure 4.5. Since a product is independent of the order, we can equally well calculate a 2-D DFT by calculating the DFT of all the columns first, then calculating the DFT of all the rows of the result.



(a) Original image (b) DFT of each row of (a) (c) DFT of each column of (b)

Figure 4.5: Calculating a 2D DFT

Linearity An important property of the DFT is its linearity; the DFT of a sum is equal to the sum of the individual DFT's, and the same goes for scalar multiplication:

$$\begin{aligned} \mathcal{F}(f+g) &= \mathcal{F}(f)+\mathcal{F}(g) \\ \mathcal{F}(kf) &= k\mathcal{F}(f) \end{aligned}$$

where k is a scalar, and f and g are matrices. This follows directly from the definition given in equation 4.4.

This property is of great use in dealing with image degradation such as noise which can be modelled as a sum:

$$d = f + n$$

where f is the original image; n is the noise, and d is the degraded image. Since

$$\mathcal{F}(d) = \mathcal{F}(f) + \mathcal{F}(n)$$

we may be able to remove or reduce n by modifying the transform. As we shall see, some noise appears on the DFT in a way which makes it particularly easy to remove.

The convolution theorem. This result provides one of the most powerful advantages of using the DFT. Suppose we wish to convolve an image M with a spatial filter S. Our method has been place S over each pixel of M in turn, calculate the product of all corresponding grey values of M and elements of S, and add the results. The result is called the *digital convolution* of M and S, and is denoted

$$M * S.$$

This method of convolution can be very slow, especially if S is large. The *convolution theorem* states that the result $M * S$ can be obtained by the following sequence of steps:

1. Pad S with zeroes so that is the same size as M; denote this padded result by S'.
2. Form the DFT's of both M and S, to obtain $\mathcal{F}(M)$ and $\mathcal{F}(S')$.
3. Form the element-by-element product of these two transforms:

$$\mathcal{F}(M) \cdot \mathcal{F}(S').$$

4. Take the inverse transform of the result:

$$\mathcal{F}^{-1}(\mathcal{F}(M) \cdot \mathcal{F}(S')).$$

Put simply, the convolution theorem states:

$$M * S = \mathcal{F}^{-1}(\mathcal{F}(M) \cdot \mathcal{F}(S'))$$

or equivalently that

$$\mathcal{F}(M * S) = \mathcal{F}(M) \cdot \mathcal{F}(S').$$

Although this might seem like an unnecessarily clumsy and roundabout way of computing something so simple as a convolution, it can have enormous speed advantages if S is large.

For example, suppose we wish to convolve a 512×512 image with a 32×32 filter. To do this directly would require $32^2 = 1024$ multiplications for each pixel, of which there are $512 \times 512 = 262144$. Thus there will be a total of $1024 \times 262144 = 268,435,456$ multiplications needed. Now look at applying the DFT (using an FFT algorithm). Each row requires 4608 multiplications by table 4.1; there are 512 rows, so a total of $4608 \times 512 = 2359296$ multiplications; the same must be done again for the columns. Thus to obtain the DFT of the image requires 4718592 multiplications. We need the same amount to obtain the DFT of the filter, and for the inverse DFT. We also require 512×512 multiplications to perform the product of the two transforms.

Thus the total number of multiplications needed to perform convolution using the DFT is

$$4718592 \times 3 + 262144 = 14,417,920$$

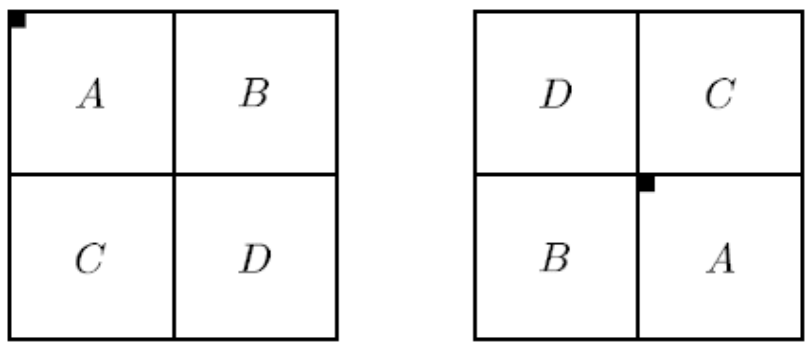
which is an enormous saving compared to the direct method.

The DC coefficient. The value $F(0,0)$ of the DFT is called the *DC coefficient*. If we put $u = v = 0$ in the definition given in equation 4.4 then

$$F(0,0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp(0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y).$$

That is, this term is equal to the sum of *all* terms in the original matrix.

Shifting. For purposes of display, it is convenient to have the DC coefficient in the centre of the matrix. This will happen if all elements $f(x,y)$ in the matrix are multiplied by $(-1)^{x+y}$ before the transform. Figure 4.6 demonstrates how the matrix is shifted by this method. In each diagram the DC coefficient is the top left hand element of submatrix A, and is shown as a black square.



An FFT After shifting

Figure 4.6: Shifting a DFT

Conjugate symmetry An analysis of the Fourier transform definition leads to a symmetry property; if we make the substitutions $u = -u$ and $v = -v$ in equation 4.4 then

$$\mathcal{F}(u, v) = \mathcal{F}^*(-u + pM, -v + qN)$$

for any integers p and q. This means that half of the transform is a mirror image of the conjugate of the other half. We can think of the top and bottom halves, or the left and right halves, being mirror images of the conjugates of each other.

Figure 4.7 demonstrates this symmetry in a shifted DFT. As with figure 4.6, the black square shows the position of the DC coefficient. The symmetry means that its information is given in just half of a transform, and the other half is redundant.

	a		a^*
b^*	B^*	d^*	A^*
	c		c^*
b	A	d	B

Figure 4.7: Conjugate symmetry in the DFT

Displaying transforms. Having obtained the Fourier transform $F(u, v)$ of an image $f(x, y)$, we would like to see what it looks like. As the elements $F(u, v)$ are complex numbers, we can't view them directly, but we can view their magnitude $|F(u, v)|$. Since these will be numbers of type `double`, generally with large range, we have two approaches

1. find the maximum value m of $|F(u, v)|$ (this will be the DC coefficient), and use `imshow` to view $|F(u, v)|/m$,

2. use `mat2gray` to view $|F(u, v)|$ directly.

One trouble is that the DC coefficient is generally very much larger than all other values. This has the effect of showing a transform as a single white dot surrounded by black. One way of stretching out the values is to take the logarithm of $|F(u, v)|$ and to display

$$\log(1 + |F(u, v)|).$$

The display of the magnitude of a Fourier transform is called the *spectrum* of the transform. We shall see some examples later on.

Fourier transforms in MATLAB

The relevant MATLAB functions for us are:

- `fft` which takes the DFT of a vector,
- `ifft` which takes the inverse DFT of a vector,
- `fft2` which takes the DFT of a matrix,
- `ifft2` which takes the inverse DFT of a matrix,
- `fftshift` which shifts a transform as shown in figure 4.6.

of which we have seen the first two above.

Before attacking a few images, let's take the Fourier transform of a few small matrices to get more of an idea what the DFT "does".

Example 1. Suppose we take a constant matrix $f(x, y) = 1$. Going back to the idea of a sum of corrugations, then *no* corrugations are required to form a constant. Thus we would hope that the DFT consists of a DC coefficient and zeroes everywhere else. We will use the `ones` function, which produces an $n \times n$ matrix consisting of 1's, where n is an input to the function.

```
>> a=ones(8);
>> fft2(a)
```

The result is indeed as we expected:

```
ans =
      64     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
```

Note that the DC coefficient is indeed the sum of all the matrix values.

Example 2. Now we'll take a matrix consisting of a single corrugation:

```
>> a = [100 200; 100 200];
>> a = repmat(a,4,4)

ans =
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200

>> af = fft2(a)

ans =

    9600     0     0     0 -3200     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
```

What we have here is really the sum of two matrices: a constant matrix each element of which is 150, and a corrugation which alternates -50 and 50 from left to right. The constant matrix alone would produce (as in example 1), a DC coefficient alone of value $64 \times 150 = 9600$; the corrugation a single value. By linearity, the DFT will consist of just the two values.

Now we shall perform the Fourier transform with a shift, to place the DC coefficient in the centre, and since it contains some complex values, for simplicity we shall just show the rounded absolute values:

```
>> af=fftshift(fft2(a));
>> round(abs(af))

ans =

     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     9     0    21    32    21     0     9
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

The DC coefficient is of course the sum of all values of `a`; the other values may be considered to be the coefficients of the necessary sine functions required to from an edge, as given in equation 4.1. The mirroring of values about the DC coefficient is a consequence of the symmetry of the DFT.

Fourier transforms of images

We shall create a few simple images, and see what the Fourier transform produces.

Example 1. We shall produce a simple image consisting of a single edge:

```
>> a=[zeros(256,128) ones(256,128)];
```

This is displayed on the left in figure 4.9. Now we shall take its DFT, and shift it:

```
>> af=fftshift(fft2(a));
```

Now we'll view its spectrum; we have the choice of two commands:

1. `afl=log(1+abs(af));`

   `imshow(afl/afl(129,129))`

   This works because after shifting, the DC coefficient is at position $x = 129$, $y = 129$. We stretch the transform using `log`, and divide the result by the middle value to obtain matrix of type `double` with values in the range 0.0–1.0. This can then be viewed directly with `imshow`.

2. `imshow(mat2gray(log(1+abs(af))))`

   The `mat2gray` function automatically scales a matrix for display as an image, as we have seen in chapter 3

It is in fact convenient to write a small function for viewing transforms. One such is shown in figure 4.8. Then for example

```
function fftshow(f,type)

% Usage:  FFTSHOW(F,TYPE)
%
% Displays the fft matrix F using imshow, where TYPE must be one of
% 'abs' or 'log'.  If TYPE='abs', then then abs(f) is displayed; if
% TYPE='log' then log(1+abs(f)) is displayed.  If TYPE is omitted, then
% 'log' is chosen as a default.
%
%  Example:
%    c=imread('cameraman.tif');
%    cf=fftshift(fft2(c));
%    fftshow(cf,'abs')
%

if nargin<2,
  type='log';
end

if (type=='log')
  fl = log(1+abs(f));
  fm = max(fl(:));
  imshow(im2uint8(fl/fm))
elseif (type=='abs')
  fa=abs(f);
  fm=max(fa(:));
  imshow(fa/fm)
else
  error('TYPE must be abs or log.');
end;
```

Figure 4.8: A function to display a Fourier transform

```
>> fftshow(af,'log')
```

will show the logarithm of the absolute values of the transform, and

```
>> fftshow(af,'abs')
```

will show the absolute values of the transform without any scaling.

The result is shown on the right in figure 4.9. We observe immediately that the result is simila

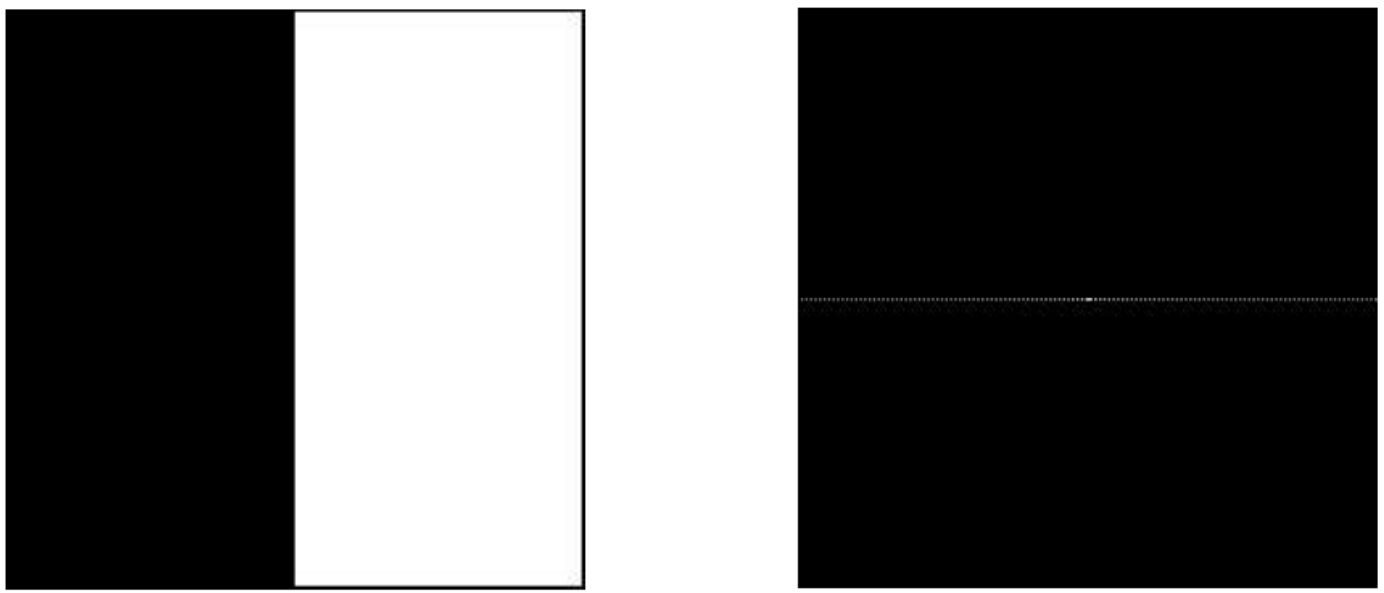


Figure 4.9: A single edge and its DFT

(although larger) to example 3 in the previous section.

Example 2. Now we'll create a box, and then its Fourier transform:

```
>> a=zeros(256,256);
>> a(78:178,78:178)=1;
>> imshow(a)
>> af=fftshift(fft2(a));
>> figure,fftshow(af,'abs')
```

The box is shown on the left in figure 4.10, and its Fourier transform is is shown on the right.

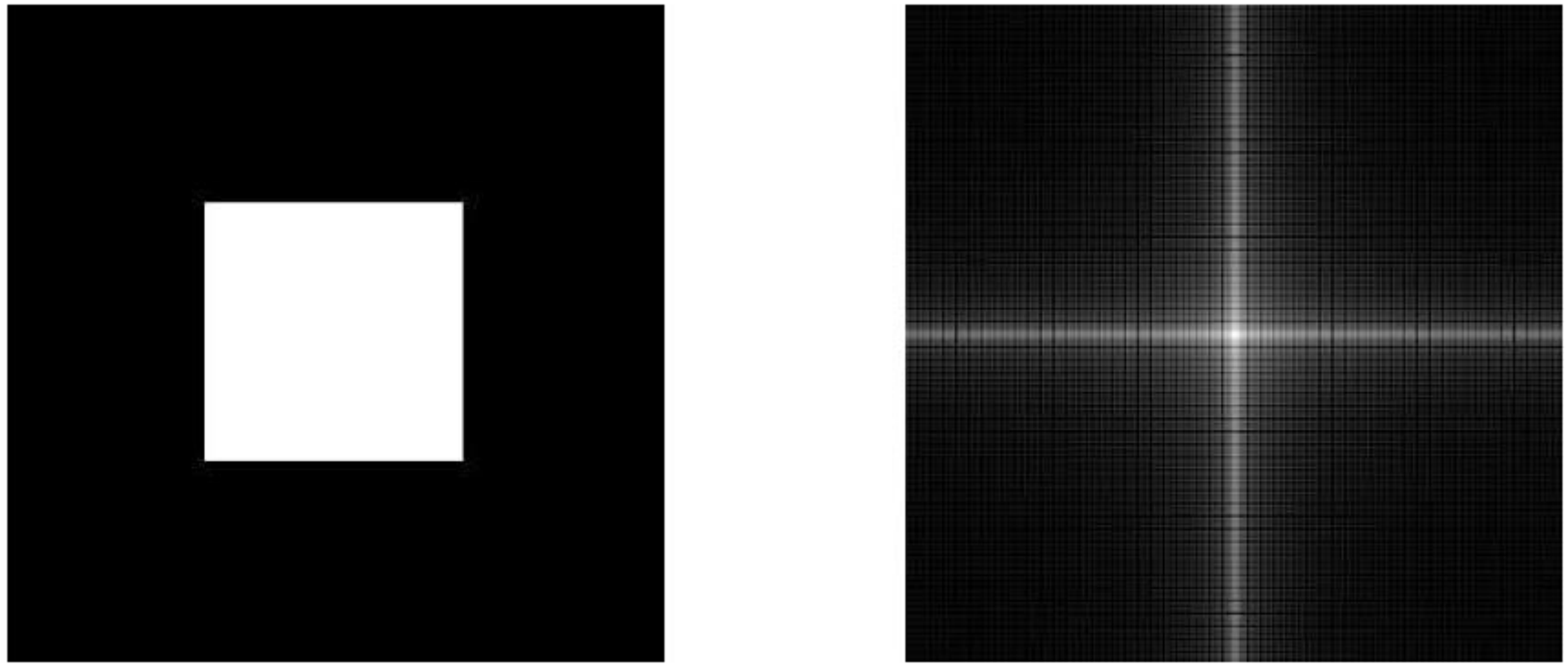


Figure 4.10: A box and its DFT

Example 3. Now we shall look at a box rotated 45°.

```
>> [x,y]=meshgrid(1:256,1:256);
>> b=(x+y<329)&(x+y>182)&(x-y>-67)&(x-y<73);
>> imshow(b)
>> bf=fftshift(fft2(b));
>> figure,fftshow(bf)
```

The results are shown in figure 4.11. Note that the transform of the rotated box is the rotated transform of the original box.

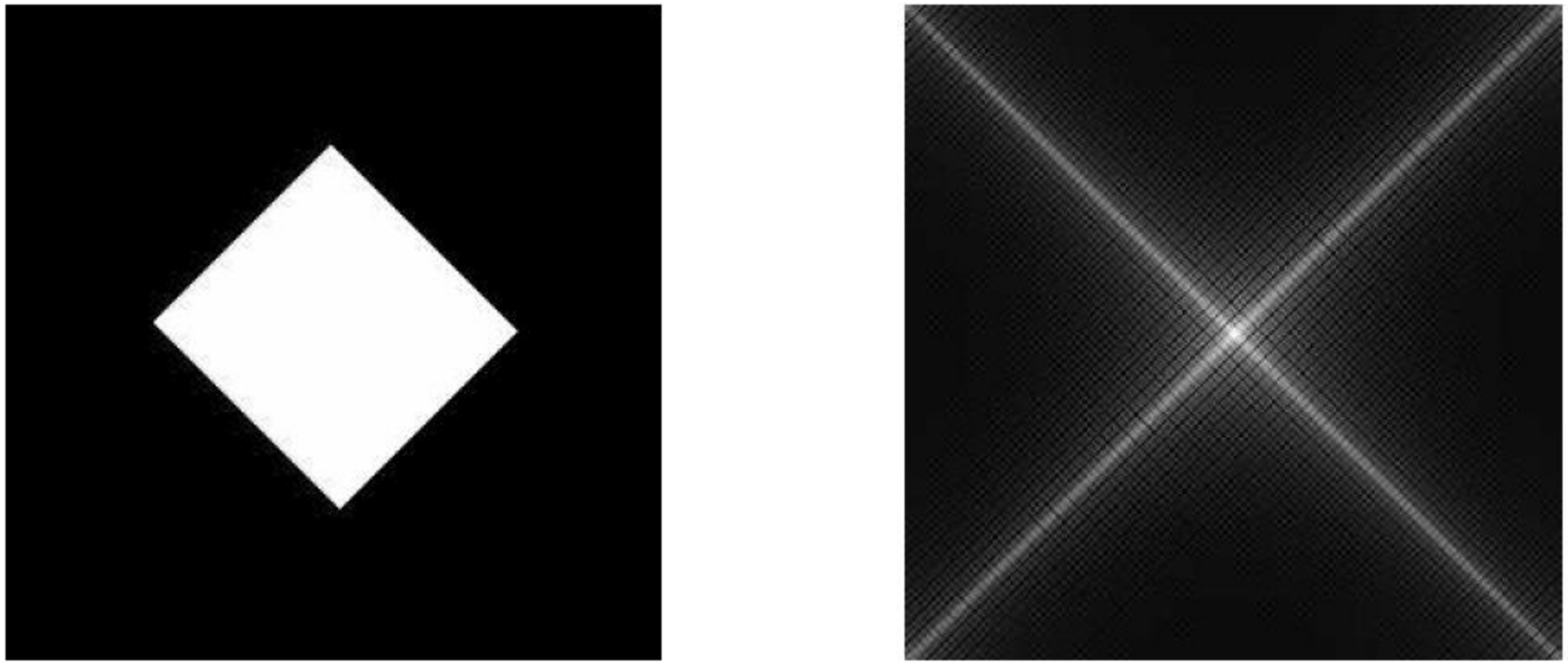


Figure 4.11: A rotated box and its DFT

Example 4. We will create a small circle, and then transform it:

```
>> [x,y]=meshgrid(-128:217,-128:127);
>> z=sqrt(x.^2+y.^2);
>> c=(z<15);
```

The result is shown on the left in figure 4.12. Now we will create its Fourier transform and display it:

```
>> cf=fft2shift(fft2(z));
>> fftshow(cf,'log')
```

and this is shown on the right in figure 4.12. Note the "ringing" in the Fourier transform. This is an

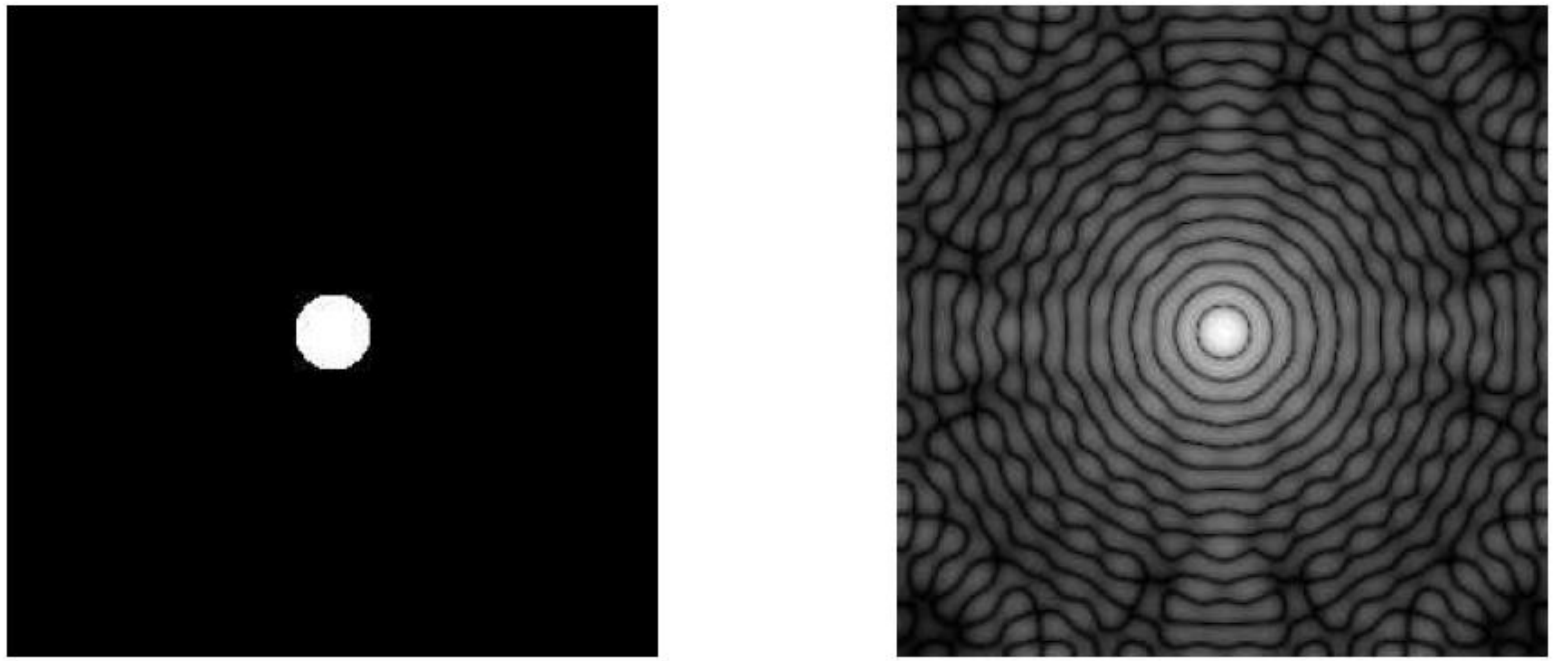


Figure 4.12: A circle and its DFT

artifact associated with the sharp cutoff of the circle. As we have seen from both the edge and box images in the previous examples, an edge appears in the transform as a line of values at right angles to the edge. We may consider the values on the line as being the coefficients of the appropriate corrugation functions which sum to the edge. With the circle, we have lines of values radiating out from the circle; these values appear as circles in the transform.

A circle with a gentle cutoff, so that its edge appears blurred, will have a transform with no ringing. Such a circle can be made with the command (given `z` above):

```
b=1./(1+(z./15).^2);
```

This image appears as a blurred circle, and its transform is very similar—check them out!

Filtering in the frequency domain

We have seen in section 4.4 that one of the reasons for the use of the Fourier transform in image processing is due to the convolution theorem: a spatial convolution can be performed by element-wise multiplication of the Fourier transform by a suitable "filter matrix". In this section we shall explore some filtering by this method.

Ideal filtering

Low pass filtering

Suppose we have a Fourier transform matrix F, shifted so that the DC coefficient is in the centre. Since the low frequency components are towards the centre, we can perform low pass filtering by multiplying the transform by a matrix in such a way that centre values are maintained, and values away from the centre are either removed or minimized. One way to do this is to multiply by an *ideal low-pass matrix*, which is a binary matrix m defined by:

$$m(x,y) = \begin{cases} 1 & \text{if } (x,y) \text{ is closer to the center than some value } D, \\ 0 & \text{if } (x,y) \text{ is further from the center than } D. \end{cases}$$

The circle `c` displayed in figure 4.12 is just such a matrix, with $D = 15$. Then the inverse Fourier transform of the element-wise product of F and m is the result we require:

$$\mathcal{F}^{-1}(F \cdot m).$$

Let's see what happens if we apply this filter to an image. First we obtain an image and its DFT.

```
>> cm=imread('cameraman.tif');
>> cf=fftshift(fft2(cm));
>> figure,fftshow(cf,'log')
```

The cameraman image and its DFT are shown in figure 4.13. Now we can perform a low pass filter

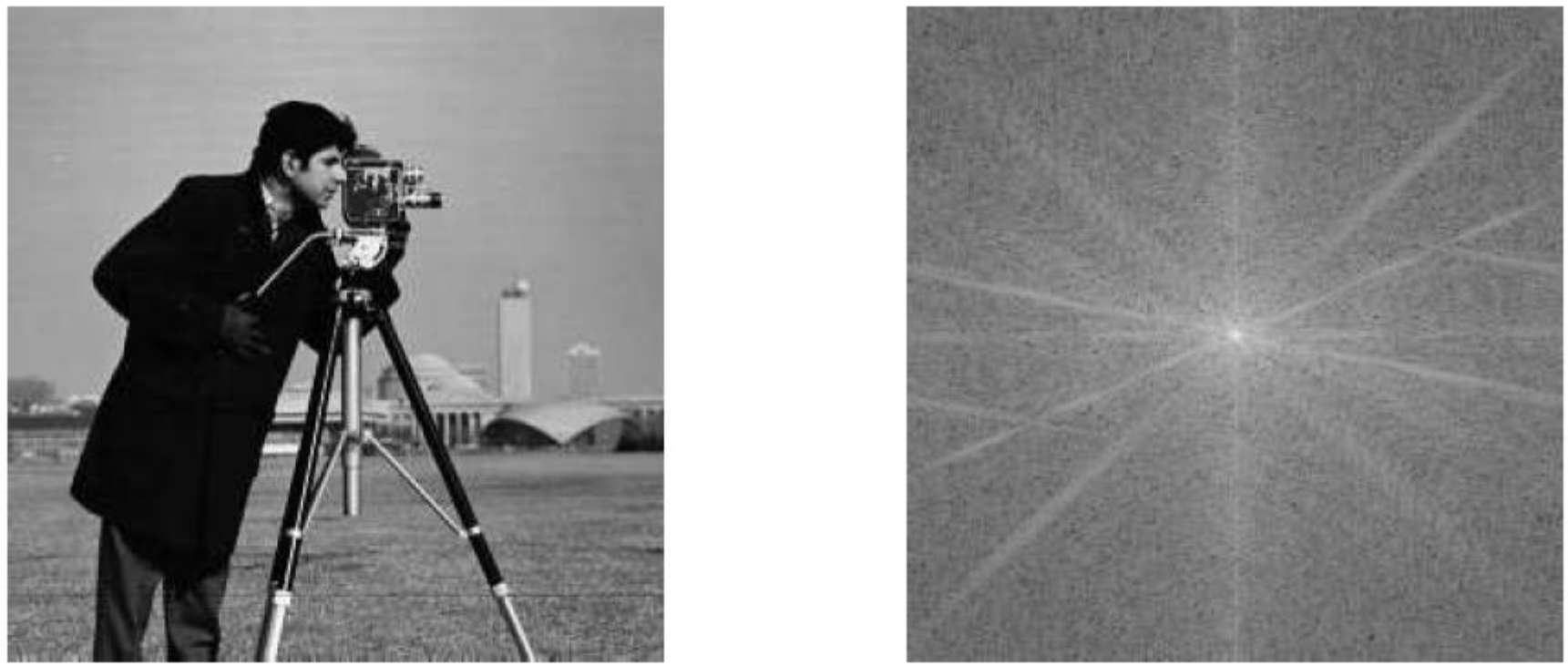


Figure 4.13: The "cameraman" image and its DFT

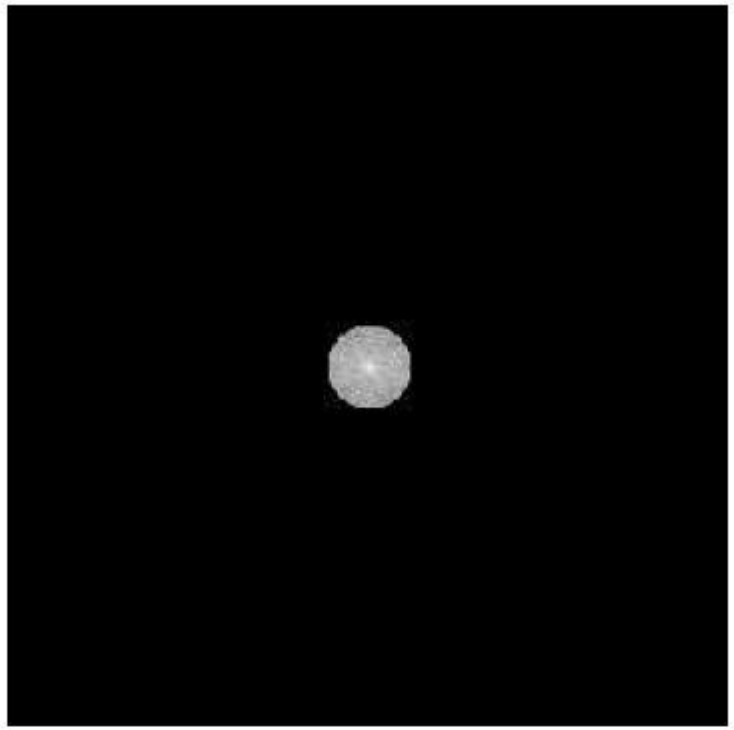
by multiplying the transform matrix by the circle matrix (recall that "dot asterisk" is the MATLAB syntax for element-wise multiplication of two matrices):

```
>> cfl=cf.*c;
>> figure,fftshow(cfl,'log')
```

and this is shown in figure 4.14(a). Now we can take the inverse transform and display the result:

```
>> cfli=ifft2(cfl);
>> figure,fftshow(cfli,'abs')
```

and this is shown in figure 4.14(b). Note that even though `cfli` is supposedly a matrix of real numbers, we are still using `fftshow` to display it. This is because the `fft2` and `fft2` functions, being numeric, will not produce mathematically perfect results, but rather very close numeric approximations. So using `fftshow` with the `'abs'` option rounds out any errors obtained during the transform and its inverse. Note the "ringing" about the edges in this image. This is a direct result of the sharp cutoff of the circle. The ringing as shown in figure 4.12 is transferred to the image.



(a) Ideal filtering on the DFT

(b) After inversion

Figure 4.14: Applying ideal low pass filtering

of the sharp cutoff of the circle. The ringing as shown in figure 4.12 is transferred to the image.

We would expect that the smaller the circle, the more blurred the image, and the larger the circle; the less blurred. Figure 4.15 demonstrates this, using cutoffs of 5 and 30. Notice that ringing is still present, and clearly visible in figure 4.15(b).

(a) Cutoff of 5

(b) Cutoff of 30

Figure 4.15: Ideal low pass filtering with different cutoffs

High pass filtering

Just as we can perform low pass filtering by keeping the centre values of the DFT and eliminating the others, so high pass filtering can be performed by the opposite: eliminating centre values and keeping the others. This can be done with a minor modification of the preceding method of low pass filtering. First we create the circle:

```
>> [x,y]=meshgrid(-128:127,-128:127);
>> z=sqrt(x.^2+y.^2);
>> c=(z>15);
```

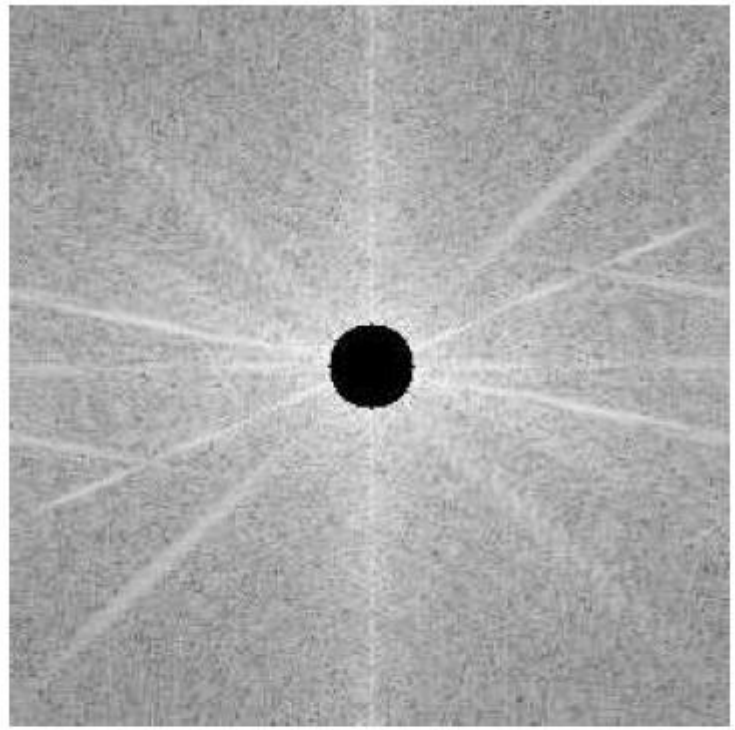
and the multiply it by the DFT of the image:

```
>> cfh=cf.*c;
>> figure,fftshow(cfh,'log')
```

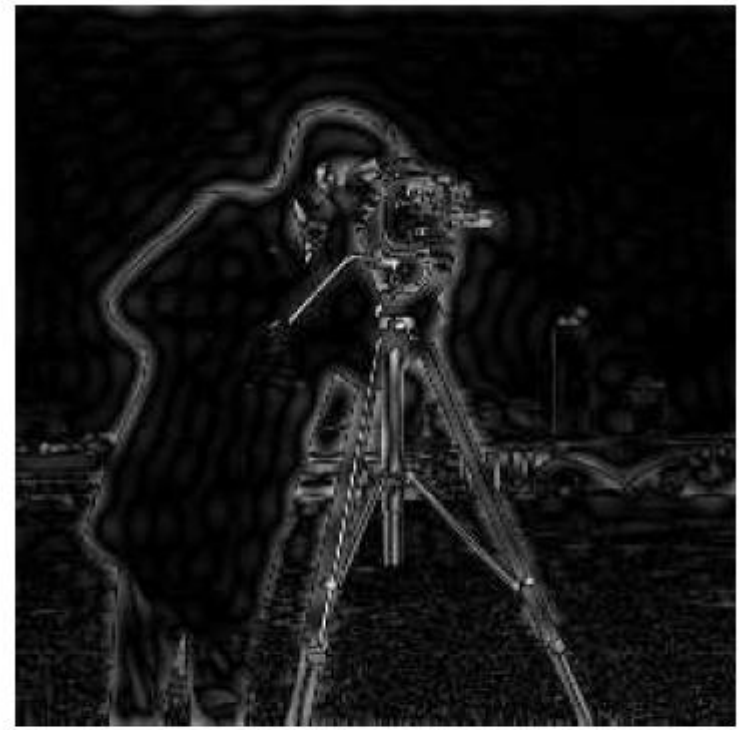
This is shown in figure 4.16(a). The inverse DFT can be easily produced and displayed:

```
>> cfhi=ifft2(cfh);
>> figure,fftshow(cfhi,'abs')
```

and this is shown in figure 4.16(b). As with low pass filtering, the size of the circle influences the information available to the inverse DFT, and hence the final result. Figure 4.17 shows some results of ideal high pass filtering with different cutoffs. If the cutoff is large, then more information is removed from the transform, leaving only the highest frequencies. This can be observed in figure 4.17(c) and (d); only the edges of the image remain. If we have small cutoff, such as in figure 4.17(a), we are only removing a small amount of the transform. We would thus expect that only the lowest frequencies of the image would be removed. And this is indeed true, as seen in figure 4.17(b); there is some greyscale detail in the final image, but large areas of low frequency are close to zero.

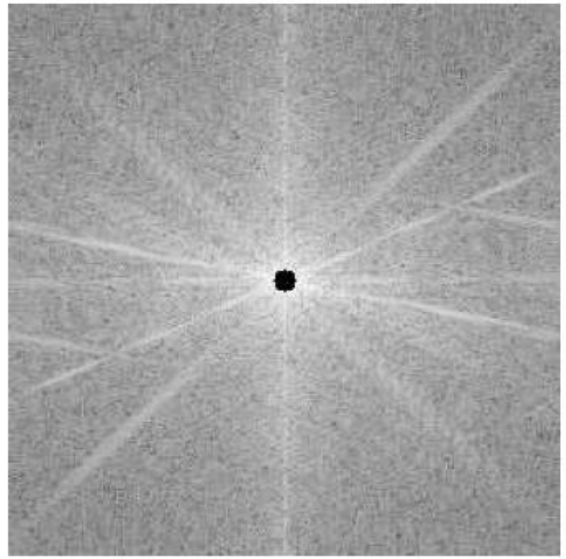


(a) The DFT after high pass filtering



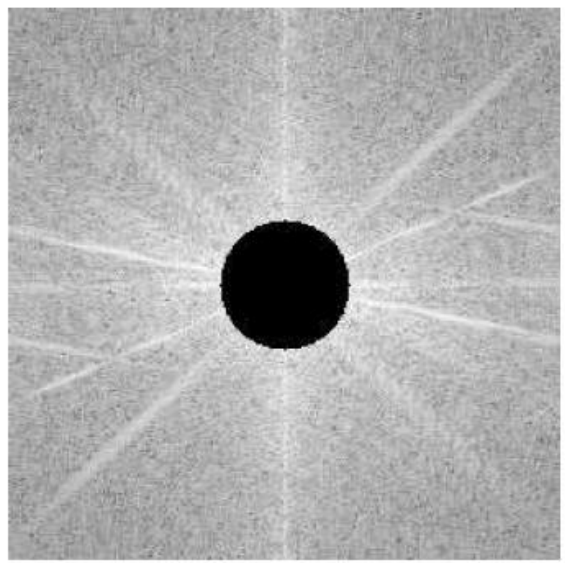
(b) The resulting image

Figure 4.16: Applying an ideal high pass filter to an image



(a) Cutoff of 5

(b) The resulting image



(a) Cutoff of 30

(b) The resulting image

Figure 4.17: Ideal high pass filtering with different cutoffs

Butterworth filtering

Ideal filtering simply cuts off the Fourier transform at some distance from the centre. This is very easy to implement, as we have seen, but has the disadvantage of introducing unwanted artifacts: ringing, into the result. One way of avoiding this is to use as a filter matrix a circle with a less sharp cutoff. A popular choice is to use *Butterworth filters*.

Before we describe these filters, we shall look again at the ideal filters. As these are radially symmetric about the centre of the transform, they can be simply described in terms of their cross sections. That is, we can describe the filter as a function of the distance x from the centre. For an ideal low pass filter, this function can be expressed as

where D is the cutoff radius. Then the ideal high pass filters can be described similarly:

$$f(x) = \begin{cases} 1 & \text{if } x > D, \\ 0 & \text{if } x \leq D \end{cases}$$

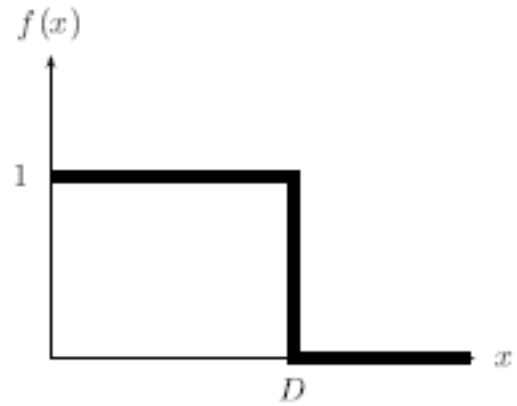
These functions are illustrated in figure 4.18. Butterworth filter functions are based on the following functions for low pass filters:

$$f(x) = \frac{1}{1 + (x/D)^{2n}}$$

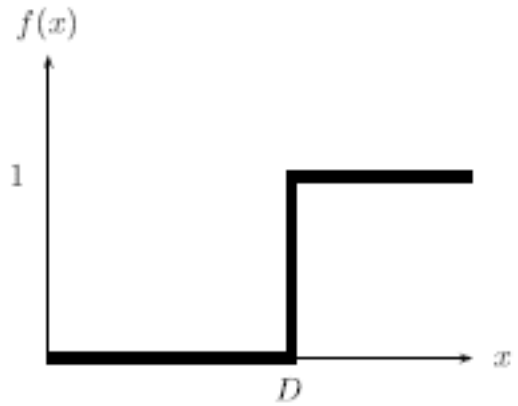
and for high pass filters:

$$f(x) = \frac{1}{1 + (D/x)^{2n}}$$

where in each case the parameter n is called the *order* of the filter. The size of n dictates the sharpness of the cutoff. These functions are illustrated in figures 4.19 and 4.20.

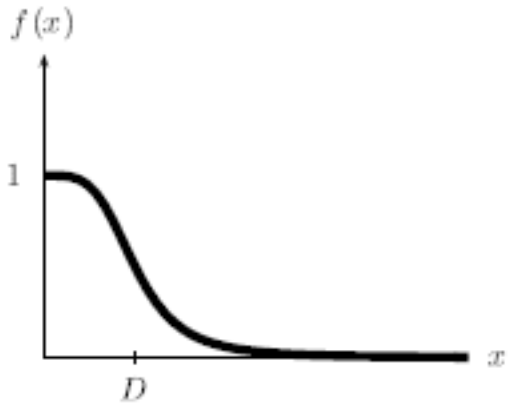


(a) Low pass

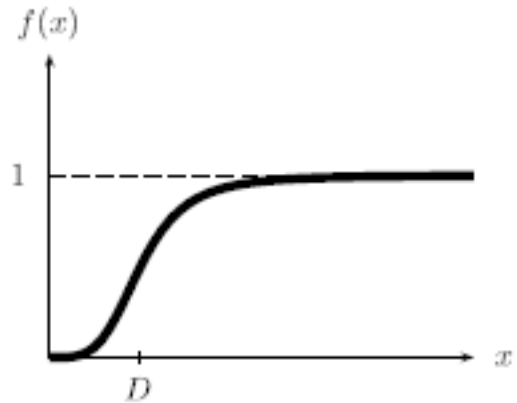


(b) High pass

Figure 4.18: Ideal filter functions

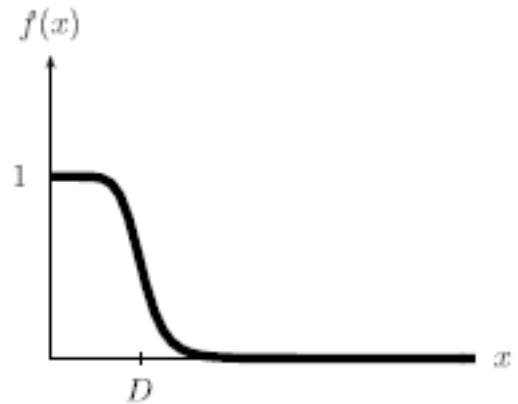


(a) Low pass

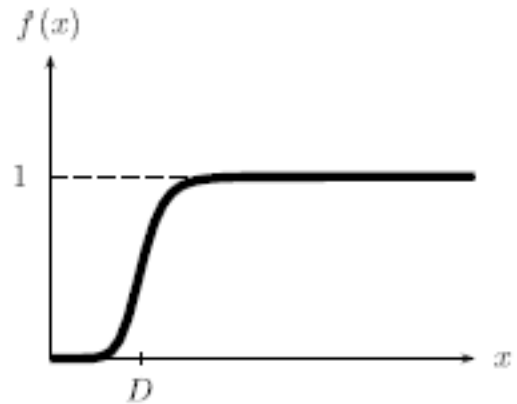


(b) High pass

Figure 4.19: Butterworth filter functions with $n = 2$



(a) Low pass



(b) High pass

Figure 4.20: Butterworth filter functions with $n = 4$

It is easy to implement these in MATLAB; here are the commands to produce a Butterworth low pass filter of size 256×256 with $D = 15$ and order $n = 2$:

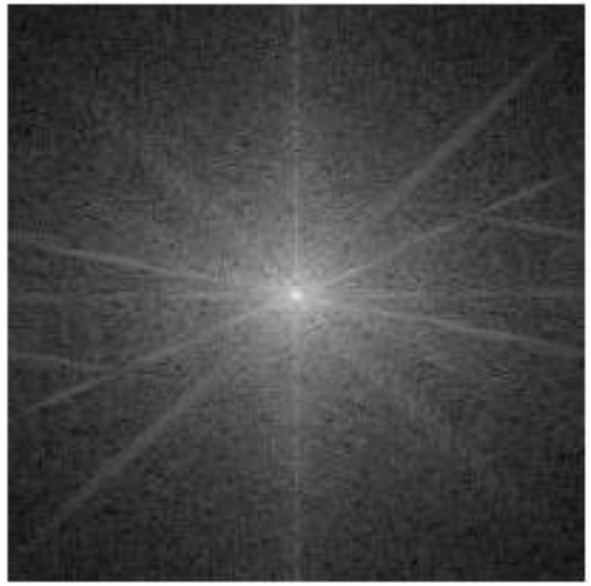
```
>> [x,y]=meshgrid(-128:217,-128:127));
>> bl=1./(1+((x.^2+y.^2)/15).^2);
```

Since a Butterworth high pass filter can be obtained by subtracting a low pass filter from 1, we can write general MATLAB functions to generate Butterworth filters of general sizes. These are shown in figures 4.21 and 4.22.

So to apply a Butterworth low pass filter to the DFT of the cameraman image:

```
>> bl=lbutter(c,15,1);
>> cfbl=cf.*bl;
>> figure,fftshow(cfbl,'log')
```

and this is shown in figure 4.23(a). Note that there is no sharp cutoff as seen in figure 4.14; also that the outer parts of the transform are not equal to zero, although they are dimmed considerably. Performing the inverse transform and displaying it as we have done previously produces figure 4.23(b). This is certainly a blurred image, but the ringing seen in figure 4.14 is completely absent. Compare the transform after multiplying with a Butterworth filter (figure 4.23(a)) with the original transform (in figure 4.13). The Butterworth filter does cause an attenuation of values away from the centre, even if they don't become suddenly zero, as with the ideal low pass filter in figure 4.14.



(a) The DFT after Butterworth low pass filtering

(b) The resulting image

Figure 4.23: Butterworth low pass filtering

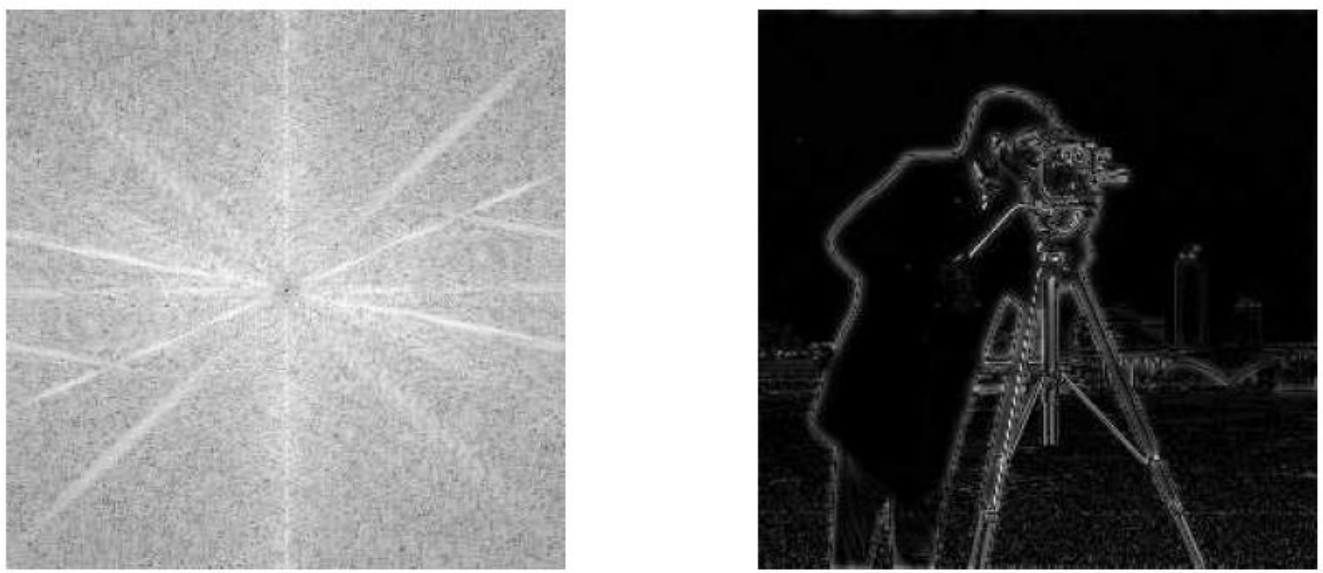
We can apply a Butterworth high pass filter similarly, first by creating the filter and applying it to the image transform:

```
>> bh=hbutter(cm,15,1);
>> cfbh=cf.*bh;
>> figure,fftshow(cfbh,'log')
```

and then inverting and displaying the result:

```
>> cfbhi=ifft2(cfbh);
>> figure,fftshow(cfbhi,'abs')
```

The images are shown in figure 4.24



(a) The DFT after Butterworth high pass filtering (b) The resulting image

Figure 4.24: Butterworth high pass filtering

Gaussian filtering

We have met Gaussian filters in chapter 3, and we saw that they could be used for low pass filtering. However, we can also use Gaussian filters in the frequency domain. As with ideal and Butterworth filters, the implementation is very simple: create a Gaussian filter, multiply it by the image transform, and invert the result. Since Gaussian filters have the very nice mathematical property that a Fourier transform of a Gaussian is a Gaussian, we should get exactly the same results as when using a linear Gaussian spatial filter.

Gaussian filters may be considered to be the most "smooth" of all the filters we have discussed so far, with ideal filters the least smooth, and Butterworth filters in the middle.

We can create Gaussian filters using the `fspecial` function, and apply them to our transform.

```
>> g1=mat2gray(fspecial('gaussian',256,10));
>> cg1=cf.*g1;
>> fftshow(cg1,'log')
>> g2=mat2gray(fspecial('gaussian',256,30));
>> cg2=cf.*g2;
>> figure,fftshow(cg2,'log')
```

Note the use of the `mat2gray` function. The `fspecial` function on its own produces a low pass Gaussian filter with a very small maximum:

```
>> g=fspecial('gaussian',256,10);
>> format long, max(g(:)), format

ans =

   0.00158757552679
```

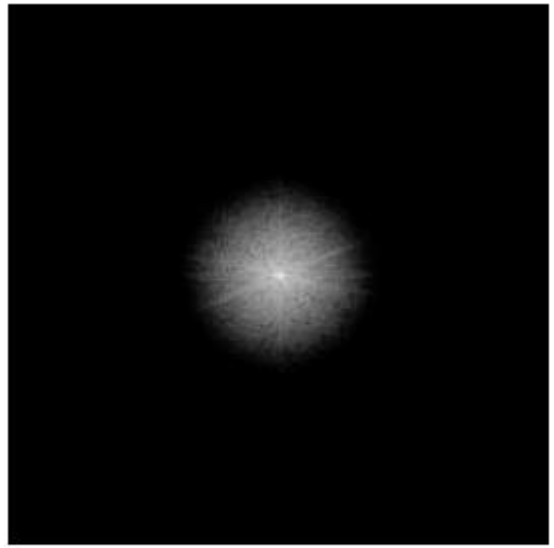
The reason is that `fspecial` adjusts its output to keep the volume under the Gaussian function always 1. This means that a wider function, with a large standard deviation, will have a low maximum. So we need to scale the result so that the central value will be 1; and `mat2gray` does that automatically.

The transforms are shown in figure 4.25(a) and (c). In each case, the final parameter of the `fspecial` function is the standard deviation; it controls the width of the filter. Clearly, the larger the standard deviation, the wider the function, and so the greater amount of the transform is preserved.

The results of the transform on the original image can be produced using the usual sequence of commands:

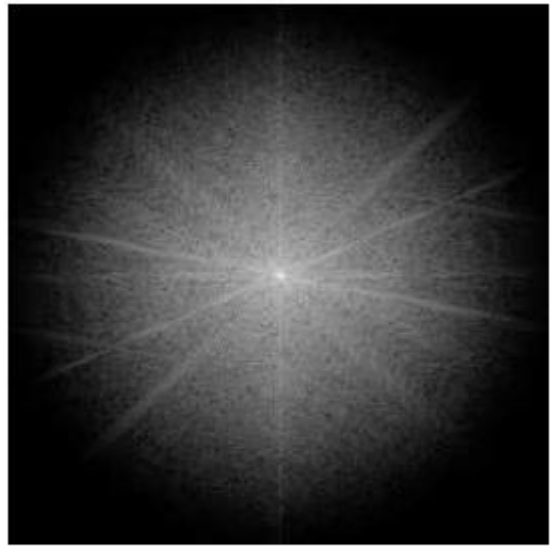
```
>> cgi1=ifft2(cg1);
>> cgi2=ifft2(cg2);
>> fftshow(cgi1,'abs');
>> fftshow(cgi2,'abs');
```

and the results are shown in figure 4.25(b) and (d)



(a) $\sigma = 10$

(b) Resulting image



(c) $\sigma = 30$

(d) Resulting image

Figure 4.25: Applying a Gaussian low pass filter in the frequency domain

We can apply a high pass Gaussian filter easily; we create a high pass filter by subtracting a low pass filter from 1.

```
>> h1=1-g1;
>> h2=1-g2;
>> ch1=cf.*h1;
>> ch2=cf.*h2;
>> ch1i=ifft2(ch1);
>> chi1=ifft2(ch1);
>> chi2=ifft2(ch2);
>> fftshow(chi1,'abs')
>> figure,fftshow(chi2,'abs')
```

and the images are shown in figure 4.26. As with ideal and Butterworth filters, the wider the high pass filter, the more of the transform we are reducing, and the less of the original image will appear in the result.

(a) Using $\sigma = 10$ (b) Using $\sigma = 30$

Figure 4.26: Applying a Gaussian high pass filter in the frequency domain