Filtering

Jason M. Kinser

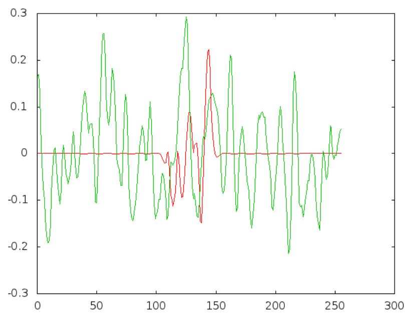
Chapter 16

Correlations

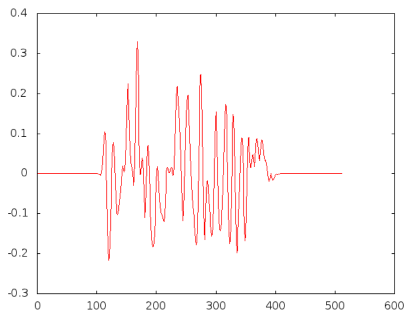
Correlation is method that compares all possible shifts of two signals. It has been useful in detection of signals embedded in noise or other clutter. This chapter will review the basic correlation theory and then advance to composite filtering. The last section describes some of the major problems with correlations.

16.1 Justification and Theory

Consider the simple example shown in Figure 16.1. In Figure 16.1(a) there are two signals. The red signal is the target signal and it is embedded somewhere in the green signal. The correlation of the two signals is shown in Figure 16.1(b) which shows a peak at $x = 168$ which is an indication that the target exists in the cluttered signal.



(a) Original signal.



(b) Correlation.

Figure 16.1: The correlation of a signal embedded in clutter.

The signals in Figure 16.1(a) are both zero-sum. The x-axis for both is 256 elements, whereas the length of the x axis in Figure 16.1(b) is double that. Every location in the output considers a shifted comparison of the target signal with the embedded signal and a peak indicates the presence of the signal as well as the location.

More formally, consider the input as a vector $\vec{f}$ and the embedded signal as $\vec{g}$. The correlation of the two is,

$$c(\omega) = \int_{-\infty}^{\infty} f(x) g^{\dagger}(x - \omega) dx, \tag{16.1}$$

where $g^{\dagger}$ represents the complex conjugate. Basically, for every shift ω the inner product is calculated. Since the signals are zero sum and there are no outlandishly large anomalies, the largest inner product occurs when the signal aligns with something very similar in $\vec{g}$. The peak in the output is at $x = 168$ which is 88 pixels from the center of the output.

Figure 16.2 shows the two signals with the input shifted 88 pixels to the left. As seen the target does exist in the signal. The difference in vertical height occurs when the signals were forced to have a zero sum. The other peaks in the correlation exist because activity in the signal is similar to the target in other locations. However, the peak of the correlation spike indicates the strength of the similarity.

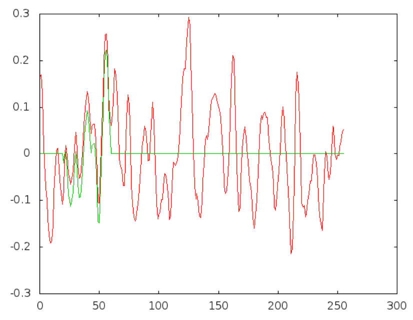


Figure 16.2: Aligning the signal with the target.

$$c(\omega, \xi) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) g^{\dagger}(x - \omega, y - \xi) dx dy, \tag{16.2}$$

where f and g are the target and signal. This operation computes the inner product of all possible shifts in two dimensions.

16.2 Brute Force Applications

The term *brute force* indicates that the algorithm is implemented in a straight forward but often slower method. An earlier example is the digital Fourier transform (see Equation **??**). This equation is easily implemented in Python but is not as efficient as the fast Fourier transform.

The *scipy.signal* module provides both correlation and convolution functions for two-dimensional data. These are good tools to use as long as the kernel is small in dimension.

Formally, the operation of correlating an image **I** with a kernel **K** is,

$$\{(\vec{x}, \mathbf{c}(\vec{x})) : \mathbf{c}(\vec{x}) = \mathbf{I}(\vec{x}) \otimes \mathbf{K}(\vec{x}), \vec{x} \in \mathbf{X}\}, \tag{16.3}$$

where $\otimes$ is a common symbol used to represent a correlation.

The first example correlates an input with a 5×5 kernel in which all nine values are 1/25. For any pixel in the input, $a_{i,j}$ the outbut $b_{i,j}$ is the average of the pixel and the neighbors, $b_{i,j} = \frac{1}{9} \sum_{m=-1}^{1} \sum_{n=-1}^{1} a_{i+m,j+n}$. This will cause the image to be slightly blurred. Increasing the size of the kernel will increase the blurring effect but will also consume more computational time. This process is demonstrated in Code 16.1 in which the input image is loaded in Line and converted to a gray scale matrix in Line 3. Line 4 creates the 5×5 block and the correlation is performed in Line 6. Figure 16.4(a) is the original input and the result of Line 6 is shown in Figure 16.4(b).

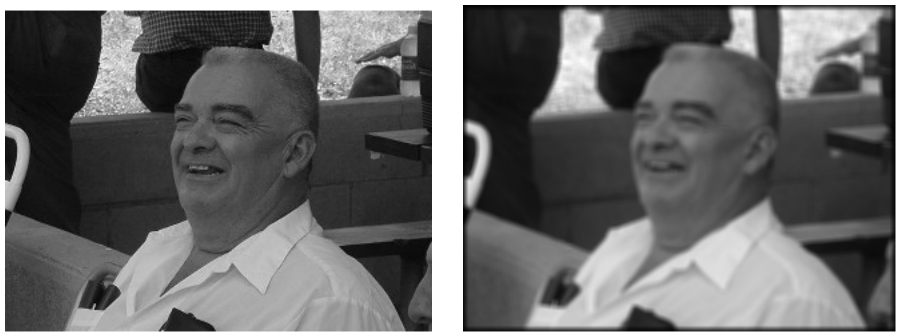
Code 16.1 Smoothing through a correlation with a small solid block.

```
1 >>> from scipy.signal import correlate2d
2 >>> mg = Image.open('man.jpg')
3 >>> orig = sophia.i2a( mg.convert('L'))/255.0
4 >>> kern = np.ones((5,5))/25.0
5 >>> corr = correlate2d( orig, kern )
```

The second example uses an *on-center/off-surround* (OCOS) filter to enhance edges. While there are many variants a very simple kernel with zero-sum is,

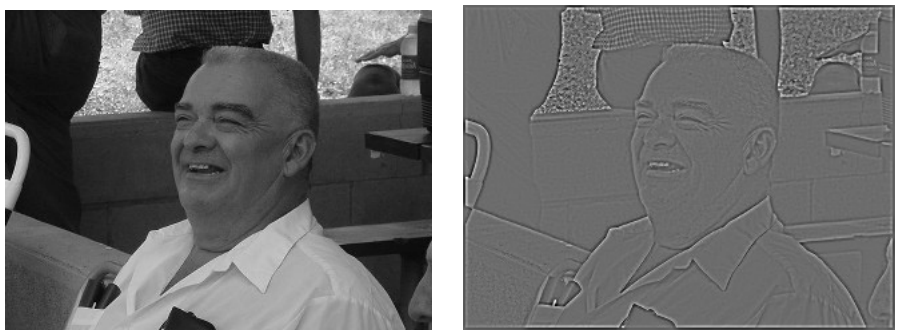
$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 16 & 0 & -1 \\ -1 & 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}.$$

The result is shown in Figure 16.4.



(a) The original input image. (b) Smoothed.

Figure 16.3: Correlation with a small solid block (Code 16.1) creates the smoothed version of the image.



(a) The original input image. (b) Edge.

Figure 16.4: Correlation with a small OCOS filter creates the edge enhanced version of the image.

16.3 Implementation

The implementation of a digital version of (16.2) is not very difficult but it would entail four nested loops. Code 16.2 shows the simple implementation where a and b represent the two images. For large images this code would be slow in any programming language. Using a scripting language like Python this code would be impossibly slow.

Code 16.2 Inefficient script for a correlation.

```
>>> c = zeros((N,N))
>>> for i in range(N):
>>>     for j in range(N):
>>>         for k in range(N):
>>>             for l in range(N):
>>>                 c[i,j] += a[i,j] * b[i-k,j-l].conjugate()
```

16.3.1 Computations in Fourier Space

The solution is to compute the correlations in Fourier space. To present the process consider again the correlation in (16.1). The Fourier transform of the correlation is,

$$\mathcal{F}\{c(\omega)\} = \int_{-\infty}^{\infty} c(\omega) \exp\left[-i\omega\alpha\right] d\alpha \tag{16.4}$$

Now define $z = x - \alpha$ and thus also $dx = -d\alpha$ and $\alpha = x - z$. Substituting these relationships and Equation (16.1) into Equation (16.4) yields,

$$\mathcal{F}\{c(\omega)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x) g^{\dagger}(z) \exp\left[-i\omega(x-z)\right] dx\ dz. \tag{16.5}$$

This can be rewritten as,

$$\mathcal{F}\{c(\omega)\} = -\int_{-\infty}^{\infty} f(x) \exp\left[-i\omega x\right] dx \int_{-\infty}^{\infty} g^{\dagger}(x) \exp\left[-i\omega z\right] dz. \tag{16.6}$$

Each of the integrals are in fact Fourier transforms,

$$\mathcal{F}\{c(\omega)\} = F(\omega) G^{\dagger}(\omega). \tag{16.7}$$

The Fourier transform of the correlation is actually the multiplication of the Fourier transforms of the two signals. For a signal of length N the digital version of Equation (16.1) would require two nested loops and so the number of operations (multiple-additions) is N^2. Recall that the FFT requires $N \log_2 N$ operations and thus the number of operations required to compute the correlation using Fourier space is $N + 3N \log_2 N$ which is a tremendous savings for large N. The savings is amplified for signals in two-dimensions.

So, the steps for computing the correlation are:

1. Compute the FFTs of the two input signals.
2. Multiply the two FFTs (with the complex conjugate applied to one of them),
3. Compute the inverse FFT.

16.3.2 Implementation in Python

A Python implementation is shown in Code 16.3. This function receives two signals and follows the previous protocol. The function **Correlate** in the *sophia* module actually has an optional switch. In some cases the data is already in the Fourier space from other processes and when the switch is set to 1 then the correlation process omits Lines 4 and 5.

Code 16.3 Using the **Correlate** function.

```
1   # sophia.py
2  def Correlate( A, B,):
3      a = fftpack.fft(A)
4      b = fftpack.fft(B)
5      c = a * b.conjugate( )
6      C = fftpack.ifft( c );
7      C = Swap(C)
8      return C
```

An example is shown in Code 16.4. The function **GeoImage** creates an image with rectangles and circles which is shown in Figure 16.5(a). The image as shown has the intensities reversed for viewing. The function **GeoTest** creates a target image which is an annular ring that has the same radius as the largest circle in Figure 16.5(a). The correlation is performed in Line 15 and the result is shown (with inverted intensities) in Figure 16.5(b).

Code 16.4 Correlating shapes.

```
# corrs.py
def GeoImage( ):
    mat = zeros( (512,512) )
    mat[100:130,80:120] = 1
    mat[300:350,100:210] = 1
    mat[80:210,300:400] = 1
    mat += sophia.Circle( (512,512), (150,180), 40 )
    mat += sophia.Circle( (512,512), (350,300), 60 )
    return mat

def GeoTest( geomat ):
    targ = zeros( (512,512) )
    targ = sophia.Circle( (512,512), (256,256), 60 )
    targ -= sophia.Circle( (512,512), (256,256), 55 )
    corr = sophia.Correlate( geomat, targ )
    return corr.real

>>> mat = GeoImage()
>>> corr = GeoTest( mat )
```

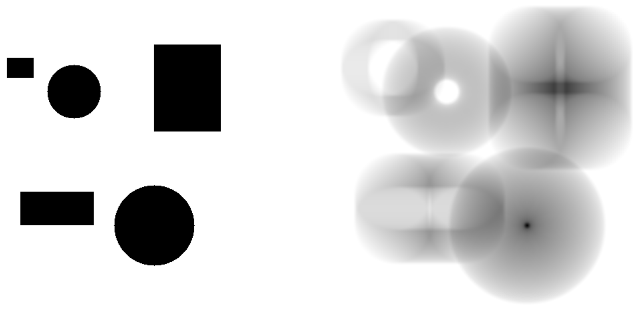

(a) Original signal. (b) Correlation.

Figure 16.5: The correlation of shapes.

It is possible to correlate with a solid circle instead of an annular ring by removing Line 14. However, the result is not as definite. The reason is that the process is merely performing a large number of inner products. Consider a large solid shape **A** and two smaller solid shapes **B** and **C** of the same area. Furthermore, **B** and **C** can completely fit inside of **A**. The inner product of **A** and **B** is the same as the inner product of **A** and **C**. Similarly, the response of the correlations will also be indistinguishable near the peak of of the correlation.

The use of the annular ring basically selects only those objects that have the same shape, (i.e., a large circle).

16.3.3 Example - Rolls Royce

Figure 16.6 shows an image of a car dashboard (might as well be a dream car) in which there are several dials that are circular in shape. Although circles towards the perimeter of the frame no longer perfectly circular. The goal of the example is to build a filter that identifies the presence and location of the dials for the tachometer and fuel gauge.

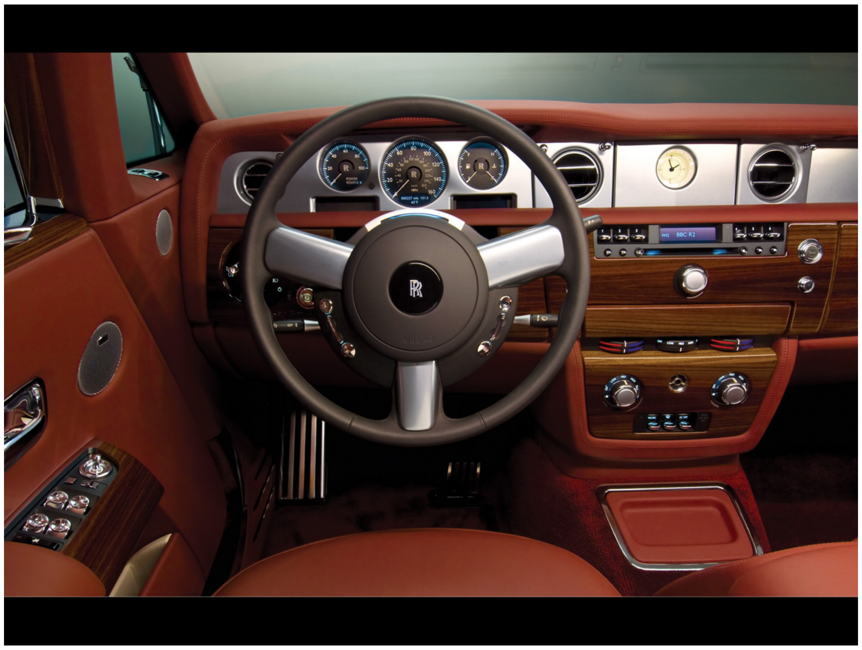


Figure 16.6: Dash of a Rolls Royce.[?]

To accomplish this task the following steps are employed:

1. Convert the image to gray scale.
2. Enhance the edges.
3. Build a filter.
4. Perform the correlation.
5. Identify the peaks.

The function **LoadRolls** function shown in Code 16.5 loads the image and converts it to a matrix in Line 4. Code 16.6 displays the **Edges** function which uses the **sobel** function to enhance the edges. The **sobel** function only enhances edges along a single axis and so it is called twice once for each axis. Line 8 combines the two results into a single edge enhanced image.

Code 16.5 The **LoadRolls** function.

```
1 # rolls.py
2 def LoadRolls( fname ):
3     mg = Image.open( fname )
4     data = sophia.i2a( mg.convert('L') )/255.0
5     return data
```

Code 16.6 The **Edges** function.

```
1 # rolls.py
2 import numpy as np
3 from scipy.ndimage import sobel
4
5 def Edges( data ):
6     edj1 = sobel( data, 0 )
7     edj2 = sobel( data, 1 )
8     edj = np.sqrt( edj1**2 + edj2**2 )
9     return edj
```

The filter is simply an annular ring which has a few pixel thickness. Since the target circle has a radius of 40 pixels the inner and outer radii are 37 and 42. Code 16.7 shows the **RingFilter** function which calls the **sophia.Circle** function to create a solid circle with a radius of 42 in Line 4. Line 5 creates a smaller circle and subtracts from the previous to generate an annular ring.

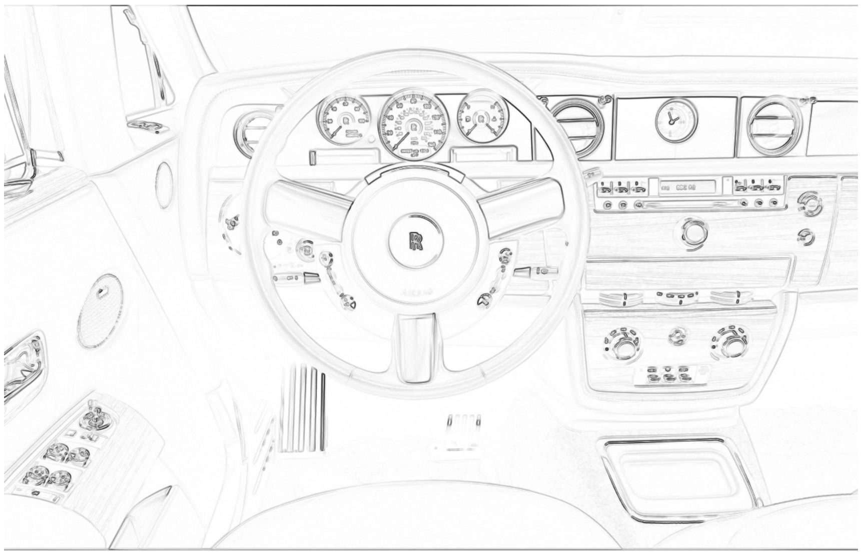


Figure 16.7: Edge enhanced version of the image.

Code 16.7 The **RingFilter** function.

```
# rolls.py
def RingFilter( vh ):
    V,H = vh
    circ = sophia.Circle( (V,H), (V/2,H/2), 42 )
    circ -= sophia.Circle( (V,H), (V/2,H/2), 37 )
    return circ
```

The **Correlate** function shown in Code 16.8 simply calls the **sophia.Correlate** function and returns the absolute value of the result. The peaks are found sequentially by iteratively finding the largest peak and then removing it and surrounding pixels from further consideration. This is accomplished in the **FindPeaks** function in Code 16.9. The correlation is shown in Figure 16.8 in which the dark pixels indicate a higher correlation energy. At the center of the identified circles there is a sharp correlation signal.

Code 16.8 The **Correlate** function.

```
# rolls.py
def Correlate( data, circ ):
    corr = sophia.Correlate( data, circ )
    corr = abs( corr )
    return corr
```

Code 16.9 The **FindPeaks** function.

```
# rolls.py
def FindPeaks( corr, N=10 ):
    work= corr + 0
    peaks = []
    V,H = corr.shape
    for i in range( N ):
        v,h = divmod( work.argmax(), H )
        peaks.append( (v,h,work[v,h]) )
        mask = 1 - sophia.Circle((V,H), (v,h), 42 )
        work *= mask
    return peaks
```

16.3.4 Example - Boat Identification

Consider the image shown in Figure 16.9 which was extracted from a much larger image. The goal is to isolate the boats. There are several methods in which this can be accomplished, but the method chosen here is to remove the dock from the image. The dock is a static structure and therefore its geometric shape can be predefined. Code

Code 16.10 shows the **BuildDock** function which constructs the dock from a set of rectangles. Line 4 creates the main artery. Lines 5 - 7 create regularly spaced

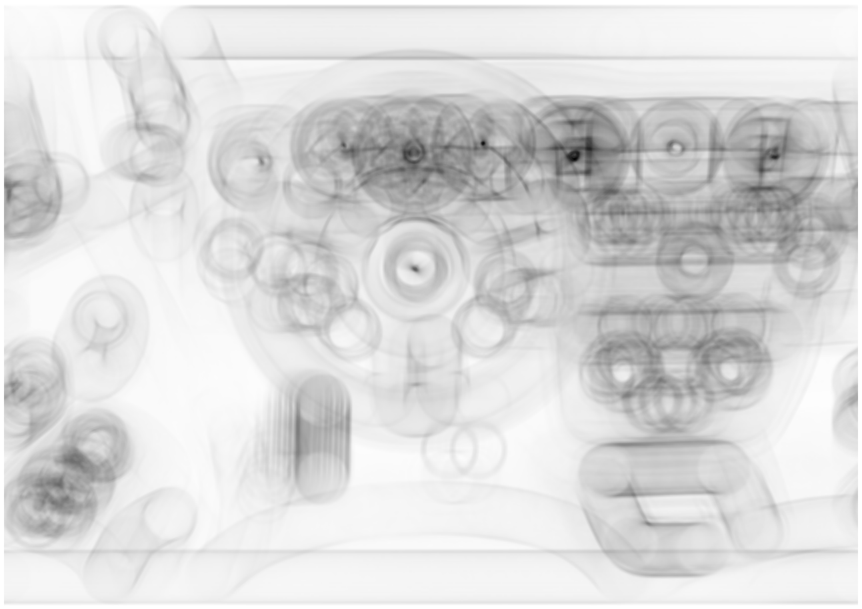


Figure 16.8: The correlation of the edge enhanced image with the filter.

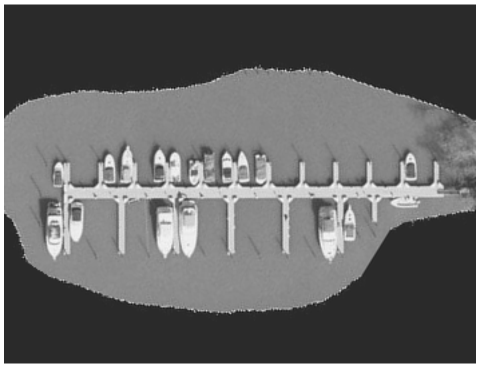


Figure 16.9: An original image extracted from a much larger image.

arms for the top portion of the dock. Lines 8 - 12 creates the arms on the lower portion. The **center_of_mass** function from the *scipy.ndimage* module computes the center of mass which is used to center the dock in the middle of the frame in Line 15. The result is shown (with inverted intensities) in Figure 16.10.

Code 16.10 The **BuildDock** function.

```
1  # fftdock.py
2  def BuildDock( ):
3      dock = np.zeros( (269,353) ) # frame size
4      dock[17:25, 0:285] = 1
5      for i in range( 12 ):
6          x = i * 25
7          dock[0:17, x:x+4] = 1
8      for i in range( 6 ):
9          x = i * 41
10         dock[25:65,x:x+4] = 1
11     i = 41*5+25
12     dock[25:43,i:i+4] = 1
13     V,H = dock.shape
14     v,h = center_of_mass( dock )
15     dock = sophia.Shift( dock, (V/2-v, H/2-h) )
16     return dock
```

The dock in the image can be eliminated if the dock mask created in **BuildDock** can be placed in the precise location. The process of locating the dock mask is performed through a correlation. This is Line 3 in Code 16.11. The peak of the correlation is found in Line 5 and this location is returned from the function.

Code 16.11 The **LocateDock** function.

```
1  # fftdock.py
2  def LocateDock( origrot, dock ):
3      corr = sophia.Correlate( origrot, dock)
4      V,H = corr.shape
5      v,h = divmod( abs(corr).argmax(), H )
6      return v,h
```

For a visual presentation the **Overlay** places the dock at location (v, h) over the original image. The result is shown in Figure **??** in which the dock mask is shown brighter than the original image. As seen the dock is correctly placed.

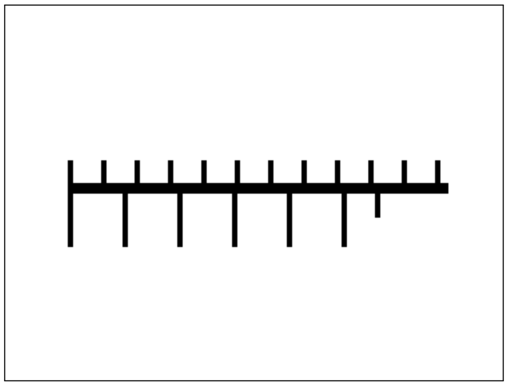


Figure 16.10: A simple construction of the dock.

Code 16.12 The **Overlay** function.

```
# fftdock.py
def Overlay( origrot, dock, vh ):
    v,h = vh
    V,H = dock.shape
    ndock = shift( dock, (v-V/2, h-H/2) ) # causes non zero values to appear
    ndock = ndock > 0.01
    return origrot + 300*ndock
```

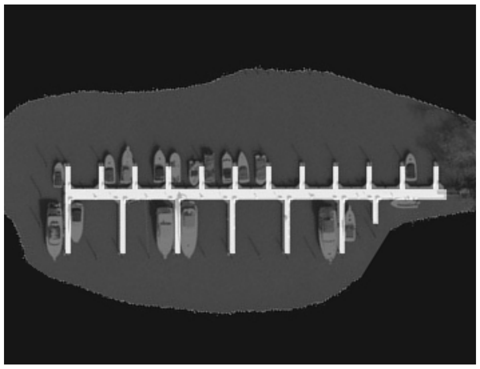


Figure 16.11: The dock mask is overlain with brighter pixels than the original to demonstrate that it is correctly located.

The subtraction of the dock is simple. However, in this case the perimeter of the dock is expanded since there are small distortions of shape in the original image. The function **SubtractDock** shown in Code 16.13 expands the dock by using the **binary_dilation** function from the *scipy.ndimage* module. Line 6 locates the and the returned image is shown in Figure 16.12.

Code 16.13 The **SubtractDock** function.

```
1 # fftdock.py
2 def SubtractDock( origrot, dock, vh ):
3     v,h = vh
4     V,H = dock.shape
5     bigdock = binary_dilation( dock, iterations=2)
6     ndock = sophia.Shift( bigdock, (v-V/2, h-H/2) )
7     ndock = ndock > 0.1
8     return origrot -200*ndock
```

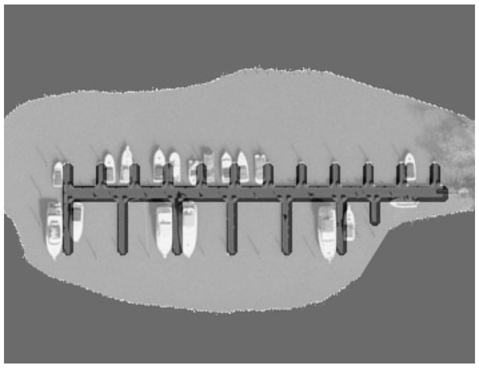


Figure 16.12: Removing the dock from the original image.

The brightest objects in the image are the boats and only the boats are the brightest objects. Recognition at this point is almost simple as shown in the function **IDboats** shown in in Code 16.14. Line 4 uses a simple threshold to separate the bright objects from the dimmer ones. Line 5 uses the **label** command from the *scipy.ndimage* module to identify islands of contiguous ON pixels. The loop started in Line 7 keeps only those islands that have at least 20 pixels such that small items are removed.

This process should isolate the boats from all else, but as seen in Figure 16.13(a) some of the boats have more than one contiguous island. These are the boats that had a darker canopy which was removed during the threshold. A vertical

Code 16.14 The **IDboats** function.

```
1  # fftdock.py
2  def IDboats( sdock ):
3      # sdock from SubtractDock
4      thsh = sdock > 140  # eliminates sea and background.
5      lbls, cnt = label( thsh )
6      answ = np.zeros( sdock.shape )
7      for i in range( 1,cnt ):
8          if (lbls==i).sum() > 20:
9              answ += (lbls==i )
10     mask = np.ones( (7,1) )
11     answ = binary_dilation( idboats, structure = mask )
12     return answ
```

structure is created in Line 10 and used in Line 11 to dilation the image but only in the vertical direction. This is controlled by the shape of `mask`. This dilation is sufficient to connect disparate parts of the boats and the final image is shown in Figure 16.13(b). In this image each boat is a single contiguous group of pixels and no other group of pixels exist. The boats are isolated.

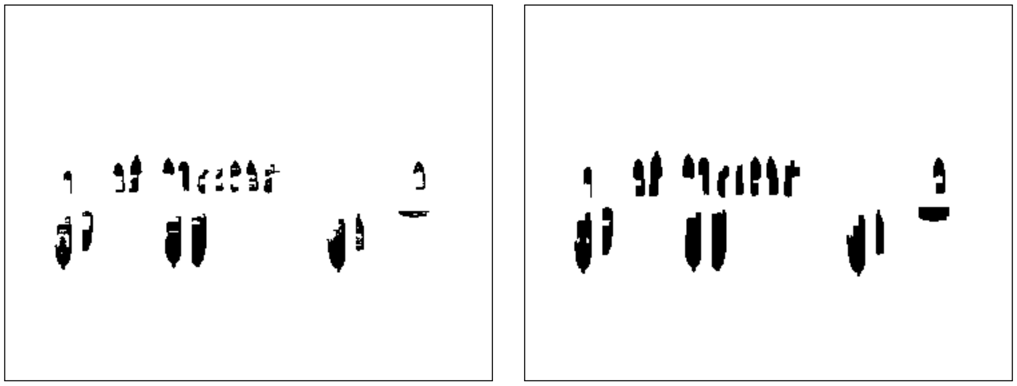


Figure 16.13: a) The boats are not isolated from the rest of the image but some boats are split into two parts. b) After dilation the boats are contiguous.

16.4 Composite Filtering

Correlation filters are somewhat useful but in many real applications the target can have several presentations. Consider the goal of building a correlation filter that would look for a horse. The horse can be of several sizes, rotations, or orientations. Furthermore, the horse can also move body parts relative to each other and comprises several poses.

The correlation filters thus constructed react to a pre-defined shape and the problem of locating a horse is far more difficult. The brute force approach would be to create a set of filters that would look for the horse in a variety of sizes, poses, orientations, and rotations. This would create a large filter back and the computation time would be too long.

16.5 SDF and MACE

One possible solution is to create a filter that can provide a desired response to several targets. The theory starts with the desire to create a filter, $\vec{h}$, that solves,

$$\vec{c} = \mathbf{X}\vec{h}, \tag{16.8}$$

where $c_i = 1 \forall i$ and the columns of $\mathbf{X}$ are the target data vectors. Extrapolation of this system to two-dimensional data is straightforward.

If $\mathbf{X}$ where square then the solution would simply require the computation of the inverse of $\mathbf{X}$. However, $\mathbf{X}$ is not square and when used for images $\mathbf{X}$ is extremely rectangular. Therefore, the pseudo-inverse is employed and the filter is computed by,

$$\vec{h} = \mathbf{X}\left(\mathbf{X}^\dagger \mathbf{X}\right)^{-1}\vec{c}. \tag{16.9}$$

The center pixel of the correlation between the filter $\vec{h}$ and any of the inputs $\vec{x}_i$ will produce c_i. This is also the center of the correlation between $\vec{h}$ and $\vec{x}_i$. Thus, $\vec{c}$ is called the *constraint vector*. The rest of the correlation surface is unconstrained.

This caused problems in applications as the side lobes of the correlation surface could be higher than the constrained correlation value. In these cases, the peak was not identifable. The solution was to simultaneously reduce the total correlation energy as well as constrain the center value. This led to the minimum average correlation energy (MACE) filter.

The MACE is computed by,

$$\hat{h} = \mathbf{D}^{-1/2}\hat{\mathbf{X}}\left(\hat{\mathbf{X}}^\dagger \mathbf{D}^{-1}\hat{\mathbf{X}}\right)^{-1}\vec{c}, \tag{16.10}$$

where,

$$D_{i,j} = \frac{\delta_{i,j}}{N} \sum_k |\hat{x}_i^{(k)}|^2, \tag{16.11}$$

where $x_i^{(k)}$ represents the i-th element of the k-th vector. The MACE must be computed with the data in Fourier space and this is designated by the carets.

The MACE filter did buld filters that produced a very sharp correlation peak when the correlation was with a training input. However, if the correlation was with the filter and another vector that was not used in training the height of the correlation peak was small. This was the case even if the correlation input vector was mostly similar to a training vector.

16.5.1 Fractional Power Filter

The original filter was robust but had large side lobes. The MACE had low side lobes and was too sensitive to changes in the input. The solution was to find a filter that was a mixture of the two.

16.5.1.1 Theory

The matrix $\mathbf{D}$ is a diagonal matrix and if the power term in Equation (16.11) where 0 instead of 2 then $\mathbf{D}$ would be the Identity matrix. In that case, Equation (16.10) would become Equation (16.9). The only difference between the two filters is the power term and thus, the FPF (fractional power filter)[?] uses Equation **??** with,

$$D_{i,j} = \frac{\delta_{i,j}}{N} \sum_k |\hat{x}_i^{(k)}|^\alpha, \tag{16.12}$$

where $0 \leq \alpha \leq 2$.

16.5.1.2 Manipulating α

The value of α ranges from 0 to 2 and when applied to images the behavior is somewhat predictable. The values in $\mathbf{D}_{i,i}$ in Equation (16.12) is proportional to the magnitude of frequency i in the training data. However, Equation (16.10) only uses the inverse of $\mathbf{D}$. Thus, the filter has high magnitudes for low magnitude frequencies in the training data. When applied to most types of images the increase in α corresponds to a reliance of the edge information in the training images. This occurs since the low frequencies are often very large and correspond to solid shapes in the input.

The example is shown in Code **??** where in Lines 4 and 5 the image from Figure 16.5(a) is opened and converted to a matrix. This image has a white background and black objects which looks good in print but is reverse of what the FPF needs. So Line 6 is used to make the background black and the targets white. It is common in filtering to ensure that the targets are the objects with higher pixel intensities.

The FPF engine considers data as vectors and so the two-dimensional image will be converted to a very long one-dimensional vector using the **ravel** command. First it is necessary to allocate space for the data. The FPF can receive multiple data vectors but in this demonstration only one data vector will be used. Nevertheless, the FPF treats data as an array of input vectors. Therefore, Line 8 allocates the space for the data vectors and in this case there is only 1 vector with $V \times H$ elements. If $\alpha > 0$ then the FPF is computed in Fourier space which necessarily has complex values. Line 10 allocates the space for the constraint vector and since there is only one training image then the length of *cst* is also 1.

Code 16.15 Computing an FPF.

```
1  >>> import Image, sophia, fpf
2  >>> from scipy.fftpack import fft2, ifft2
3  >>> from numpy import ones, zeros
4  >>> mg = Image.open( 'geos.png')
5  >>> data = sophia.i2a( mg )/255.0
6  >>> data = 1 - data
7  >>> V,H = data.shape
8  >>> X = zeros( (1,V*H), complex )
9  >>> X[0] = fft2( data ).ravel()
10 >>> cst = ones(1)
11 >>> filt = fpf.FPF( X, cst, 0 )
12 >>> filt = ifft2( filt.reshape((V,H)))
13 >>> sophia.a2i( filt.real ).show()
```

The FPF theory creates a vector **X** in which the data vectors are stored as columns. The **X** created in Line 8 contains data as rows. This is designed as a convenience for Python programmers. Within the FPF code the **X** is transposed to coincide with the theory. Line 11 calls the **fpf.FPF** function which is shown in Code 16.16. This receives the data, the constraint vector, and α.

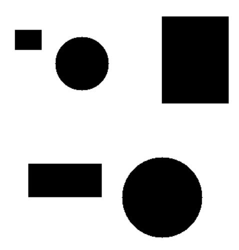
The vector returned is one-dimensional and so it is necessary to convert it back to the two-dimensional form with the **reshape** function in Line 12. This Line also converts the filter back to image space from the Fourier space.

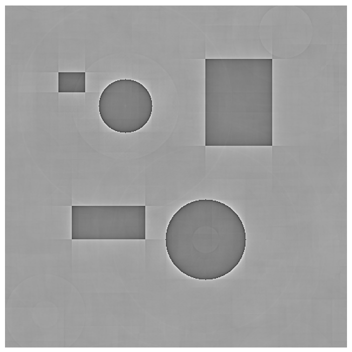
The results for two values of α are shown in Figure 16.14. Figure 16.14(a)

Code 16.16 The **FPF** function.

```
# fpf.py
def FPF( data, c, fp):
    (N,Dim )= data.shape
    D = ( pow( abs( data), fp )).sum(0)
    D = D / N
    ndx = (abs(D) < 0.001 ).nonzero()[0]
    D[ndx] = 0.001 * sign(D[ndx]+1e9)
    Y = data / sqrt( D )
    Y = Y.transpose()
    Yc = Y.conjugate().transpose()
    Q = dot( Yc, Y )        # inner product
    Rc = dot( Q,c)
    H = dot( Y, Rc ) / sqrt(D)
    return H
```

shows the filter with $\alpha = 0$ which looks exactly like the input. When $\alpha = 0$ the FPF is a weighted linear combination of the training inputs according to Equation (16.8). Figure 16.14(b) shows the FPF for $\alpha = 0.3$. This image only shows the real values of the complex valued filter and in this case there are negative numbers. Thus, all of the gray values appear to the be shifted. The gray background corresponds to pixel values of 0. The darker pixels are negative values.


(a) $\alpha = 0$.


(b) Real component from $\alpha = 0.3$.

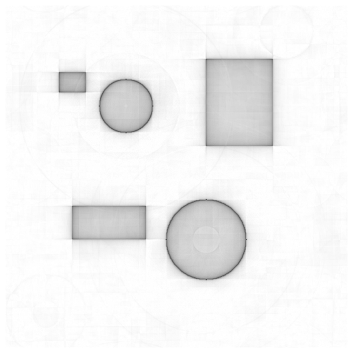

(c) Absolute value of $\alpha = 0.3$.

Figure 16.14: FPFs with different α values.

The absolute value of the same filter is shown in Figure 16.14(c) which is more conducive for displaying the effect of α. As α increases the FPF extracts the edge information and decays the pixel values in the interior. When $\alpha = 2$ the filters are generally only edges of the original shapes. An intermediate value of α as shown in this case does show a heightened sensitivity to the edges but has not completely disposed of the interior information. Thus, by manipulating α it is possible to manipulate the trade off between generalization (interior) and discrimination (edges) that is inherent in first order filters.

16.5.1.3 Example

Figure 16.15 shows the image of a tachometer with nine digits. This example creates a single filter that will recognize all nine of the numerals. The steps are:

- Load the image, convert to gray scale, and invert so the targets are brighter than the background.
- Manually locate the center of all ten numerals.
- Cut out all ten numerals.
- Create an FPF filter with $\alpha = 0.8$.
- Correlate and manually display the ten peaks.

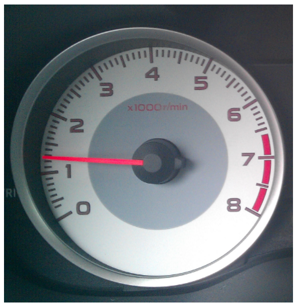


Figure 16.15: An image of a tachometer.

Code 16.17 contains two functions for loading the data and extracting the targets. The **LoadTach** function loads the image and converts it to gray scale. Line 5 inverts the data so that the targets are brighter than the background. The **Cutouts** function uses manually identified locations of the targets to extract the nine targets.

The filter is created by the **MakeTachFPF** function shown in Code 16.18. Each row vector in `X` is the FFT of each training image. These are created by

Code 16.17 The **LoadTach** and **Cutouts** functions.

```
# tachfpf.py
def LoadTach( fname ):
    mg = Image.open( fname )
    data = sophia.i2a( mg.convert('L') )/255.0
    data = 1 - data
    return data

def Cutouts( data ):
    cuts = []
    vhs = [(303,116), (250, 96), (182,107), (126,150), \
           (104,222), (116,294), (165,343), (231,362), \
           (300,341)]
    for v,h in vhs:
        cuts.append( data[v-14:v+14, h-14:h+14] + 0 )
    return cuts
```

creating an array that is the same size as the input (Line 7) and inserting the small cutout data in its center (Line 9). This is converted to Fourier space (Line 10) and rearranged into a single long vector and inserted into a row in X (Line 11). All of the elements in the constraint vector are set to 1 in Line 12 and the filter is created in Line 13 with $\alpha = 0.8$.

Code 16.19 shows the commands to run the program. Line 1 loads the data and Line 2 extracts the targets. Line 3 creates the FPF filter and Line 4 uses it to correlate with the original image. The result is shown in Figure 16.16.

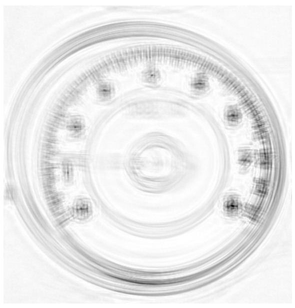


Figure 16.16: Darker pixels indicate a larger correlation response.

The result is not perfect. As seen in Figure 16.16 there are large correlation signals at the location of all nine targets. However, there are other spurious locations

Code 16.18 The **MakeTachFPF** function.

```
# tachfpf.py
def MakeTachFPF( cuts, VH ):
    V, H = VH
    NC = len( cuts )
    X = np.zeros( (NC,V*H), complex )
    for i in range( NC ):
        targ = np.zeros( VH )
        vc, hc = cuts[i].shape
        targ[V/2-vc/2:V/2+vc/2, H/2-hc/2:H/2+hc/2] = cuts[i] + 0
        targ = fft2( targ )
        X[i] = targ.ravel()
    cst = np.ones( NC )
    filt = fpf.FPF( X, cst, 0.8 )
    filt = ifft2( filt.reshape( VH ) )
    filt *= V*H
    return filt
```

Code 16.19 Running the commands to create an FPF and use it to identify the targets.

```
>>> data = LoadTach( fname )
>>> cuts = Cutouts( data )
>>> filt = MakeTachFPF( cuts, data.shape )
>>> corr = sophia.Correlate( data, filt )
>>> corr = abs( corr )
>>> sophia.a2i( corr ).show()
```

in which there is also a large correlation signal. These would be classified as false positives.

One approach for the removal of these spurious locations would be to use these locations in the training process with the corresponding constraint value set to 0 instead of 1. If, for example, a spurious correlation peak is located at (v, h) then a tenth training cutout image is extracted from this location. The constraint vector would then be (1, 1, 1, 1, 1, 1, 1, 1, 1, 0) in which the first nine elements are the same as the original training targets and the last element is the anti-training constraint for the spurious location. This new filter will produce a value of 0 in the correlation surface at (v, h). The values near this location are not constrained directly but often the constraint of one pixel drastically affects the correlation response of its neighboring pixels.

16.5.1.4 The Constraints

It is possible to set the values in $\vec{c}$ to values other than 1. One useful possibility is to set some values to 0 which creates a filter that can identify some of the inputs and reject others.

Another useful option is to set the values of the constraint vector to complex values which have an absolute value of 1. In this case, the filter has the ability to also indicate which input vector is being used as the query.

16.5.1.5 Dual FPFs

The FPF constrains the correlation peak value for the training images but not for intermediate images. Consider a case in which there are several images, $\{I_n; n \in 1, ..., N\}$ where n is a parameter such as rotation, scale, etc. The set of images are generated from a single image through the alteration of a parameter. The FPF is trained on selected values of n. Figure 16.17 shows a graph for three different values of α. The horizontal axis corresponds to n and the vertical axis corresponds to the peak correlation value. In this example there are four selected training images designated by vertical lines. The constraint vector controls the peak correlation value. The intermediate values of n also produce a peak correlation value but the peak value is lower for values of n that are farther away from the training values. This produces a "telephone pole" effect in which the peak correlation value rises nearer the training images and falls farther away from them. There are three examples shown and there is more sag in the plots for larger values of α. Since larger values of α are more discriminatory the correlation response is degraded faster as the input image differs more from the training images.

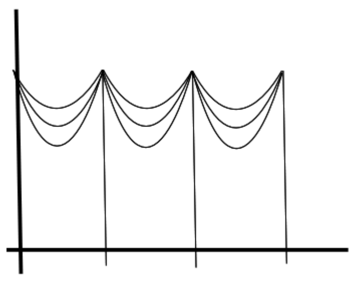


Figure 16.17: Response of the FPF for multiple values of α and for intermediate images.

The peak correlation value for several values of n can produce the same magnitude. This is depicted in Figure 16.18 where the horizontal line (constant correlation peak value) crosses the FPF response curve in multiple places. Thus, if an image with an undetermined value of n is correlated with the FPF it is not possible to determine the value of n, but it is possible to determine a set of possible values for n. In this example there are six such possible values.

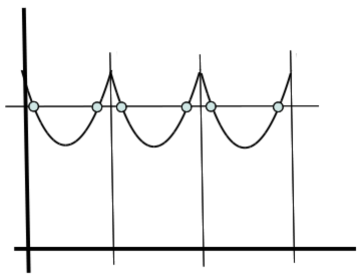


Figure 16.18: There exist multiple values of n for each correlation value.

To create a system that can determine the value of n for an input image, two FPFs are used. They are created with different training images and possibly different values of α. A real case using rotation as the parameter for n is shown in Figure 16.19. There are two curves which plot the correlation values for an image that rotates in n. The first filter is trained from a set of images with a large steps between values of n and the second is trained on a set of data with values of n that are closer together. The selection of the training images is not strict but the goal is that the two filters should react differently for all values of n. Excepting $n = 0$ the filters do not share the smae trainers or the same difference between two consecutive values of n.

Now, given an image with an unknown value of n, it is correlated with the two

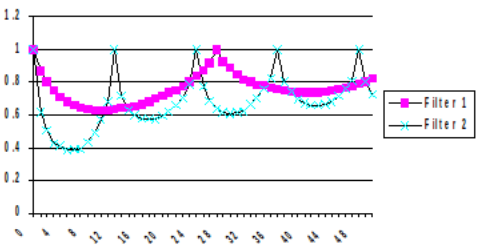


Figure 16.19: Several correlation values for an image varying in n with two different FPFs.

filters. Each provides a response and each provides a subset of possible values of n. However, if the filters are designed correctly there exists only one value of n that is in both sets of possible values. The intersection of these two sets produces a single value and this value is the estimate of n.

16.6 Restrictions of Correlations

The FPF is a good tool but it does have limitations which are common amongst correlation filters.

16.6.1 First Order

Correlation filters are inherently first order. This means that the filter responds to matches with the input vector. While it is possible to have the filter respond to a particular shape (such as the circle example) it is very difficult to build a filter that will respond to "a circle that does not have a rectangle nearby." In this latter case the problem is second order and would need a stronger tool.

16.6.2 Rotation and Scale

Correlation filters are translation invariant. In the case of the circle filter, it doesn't matter where the circle is located in the frame. The correlation spike will be located wherever the target is located.

However, there are variances that cause performance decay in the filters. Consider the case of searching for a rectangle of a known ratio. A filter could be constructed to identify this rectangle, but if the query image has a rectangle of a different scale or a different rotation then the filter may not provide a good response. The correlation filters are not rotation invariant or scale invariant.

There are a few solutions that have been proposed. One option is to construct an FPF with several rotations and scales of the target. Another solution is to transform the data into a different space. One such transformation is the polar transformation reviewed in Section **??**. The function **sophia.RPolar** converts the data from rectilinear coordinates to polar coordinates.

The advantage of this transformation is that the scale and rotation now become linear shifts in the new space. Since the filters are translation invariant, filters built in the polar space are rotation and scale invariant. The disadvantage of this conversion is that the center of the object must be known, and this information may not be available in some applications.

16.7 Operator Notation

The carat symbol is used to indicate that an image or vector is in Fourier space. Thus, an image is $\mathbf{a}[\vec{x}]$ is the image and the Fourier transform of the image is $\hat{\mathbf{a}}[\vec{x}]$. Thus,

$$\hat{\mathbf{y}}[\vec{\omega}] = \mathfrak{F}\mathbf{a}[\vec{x}],$$

where the $\vec{\omega}$ is the vector that spaces the Fourier space, $\omega \in \Omega$.

The correlation is the inverse Fourier transform of the elemental multiplication of two Fourier space images. Given the two input images $\mathbf{a}[\vec{x}]$ and $\mathbf{b}[\vec{x}]$ the correlation $\mathbf{c}[\vec{x}]$ is,

$$\mathbf{c}[\vec{x}] = \mathbf{a}[\vec{x}] \otimes \mathbf{b}[\vec{x}] = \mathfrak{F}^{-1}\left(\mathfrak{F}\mathbf{a}[\vec{x}] \times (\mathfrak{F}\mathbf{b}[\vec{x}])^{\dagger}\right)$$

where the $\dagger$ represents the complex conjugate. Since correlations are common the the $\otimes$ symbol is used to represent the operation.

Problems

Bibliography

Index