9df35ce36ec13429fe1548b8faa1bac1
ebrary

5
SciPy for Signal Processing

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We define a signal as data that measures either time-varying or spatially varying phenomena. Sound or electrocardiograms are excellent examples of time-varying quantities, while images embody the quintessential spatially varying cases. Moving images are treated with the techniques of both types of signal, obviously.

The field of signal processing treats four aspects of this kind of data – its acquisition, quality improvement, compression, and feature extraction. SciPy has many routines to treat effectively tasks in any of the four fields. All these are included in two low-level modules (`scipy.signal` being the main one, with an emphasis in time-varying data, and `scipy.ndimage`, for images). Many of the routines in these two modules are based on Discrete Fourier Transform of the data. SciPy has an extensive package of applications and definitions of these background algorithms – `scipy.fftpack`, which we will start covering first.

Discrete Fourier Transforms

The **Discrete Fourier Transform** (**DFT** from now on) transforms any signal from its time/space domain into a related signal in frequency domain. This allows us not only to be able to analyze the different frequencies of the data, but also faster filtering operations, when used properly. It is possible to turn a signal in frequency domain back to its time/spatial domain; thanks to the Inverse Fourier Transform. We will not go into detail of the mathematics behind these operators, since we assume familiarity at some level with this theory. We will focus on syntax and applications instead.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 92
http://site.ebrary.com/id/10672339?ppg=92
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

The basic routines in the `scipy.fftpack` module compute the DFT and its inverse, for discrete signals in any dimension – `fft`, `ifft` (one dimension); `fft2`, `ifft2` (two dimensions); `fftn`, `ifftn` (any number of dimensions). All of these routines assume that the data is complex valued. If we know beforehand that a particular dataset is actually real valued, and should offer real-valued frequencies, we use `rfft` and `irfft` instead, for a faster algorithm. All these routines are designed so that composition with their inverses always yields the identity. The syntax is the same in all cases, as follows:

```
fft(x[, n, axis, overwrite_x])
```

The first parameter, `x`, is always the signal in any array-like form. Note that `fft` performs one-dimensional transforms. This means in particular, that if `x` happens to be two-dimensional for example, `fft` will output another two-dimensional array where each row is the transform of each row of the original. We can change it to columns instead, with the optional parameter, `axis`. The rest of parameters are also optional; `n` indicates the length of the transform, and `overwrite_x` gets rid of the original data to save memory and resources. We usually play with the integer `n` when we need to pad the signal with zeros, or truncate it. For higher dimension, `n` is substituted by `shape` (a tuple), and `axis` by `axes` (another tuple).

9df35ce36ec13429fe1548b8faa1bac1
ebrary

To better understand the output, it is often useful to shift the zero frequencies to the center of the output arrays with `fftshift`. The inverse of this operation, `ifftshift`, is also included in the module. The following code shows some of these routines in action, when applied to a checkerboard image:

```
from scipy.fftpack import fft,fft2, fftshift
import matplotlib.pyplot as plt
B=numpy.ones((4,4)); W=numpy.zeros((4,4))
signal = numpy.bmat("B,W;W,B")
onedimfft = fft(signal,n=16)
twodimfft = fft2(signal,shape=(16,16))
plt.figure()
plt.gray()
plt.subplot(121,aspect='equal')
plt.pcolormesh(onedimfft.real)
plt.colorbar(orientation='horizontal')
plt.subplot(122,aspect='equal')
plt.pcolormesh(fftshift(twodimfft.real))
plt.colorbar(orientation='horizontal')
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

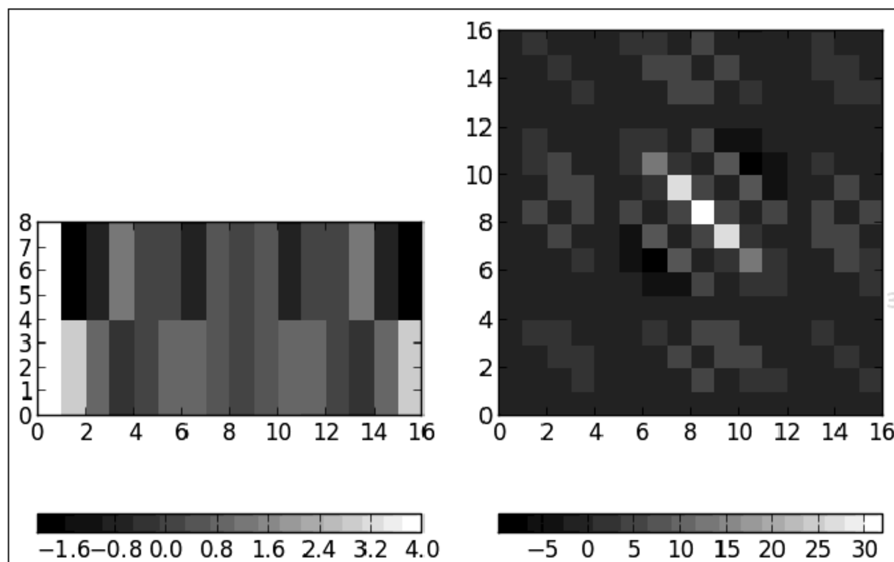
Note how the first four rows of the one-dimensional transform are equal (and so are the last four), while the two-dimensional transform (once shifted) presents a peak at the origin, and nice symmetries in the frequency domain.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 93
http://site.ebrary.com/id/10672339?ppg=93
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

In the following screenshot, the left-hand side image is `fft` and right one is `fft2` of a 2 x 2 checkerboard signal:



3429fe1548b8faa1bac1
ebrary

The `scipy.fftpack` module also offers the Discrete Cosine Transform with its inverse (`dct`, `idct`) as well as many differential and pseudo-differential operators defined in terms of all these transforms – `diff` (for derivative/integral); `hilbert`, `ihilbert` (for the Hilbert transform); `tilbert`, `itilbert` (for the h-Tilbert transform of periodic sequences); and so on.

Signal construction

9df35ce36ec13429fe1548b8faa1bac1
ebrary

To aid in the construction of signals with predetermined properties, the `scipy.signal` module has a nice collection of the most frequent one-dimensional waveforms in the literature – `chirp` and `sweep_poly` (for the frequency-swept cosine generator), `gausspulse` (a Gaussian modulated sinusoid), `sawtooth` and `square` (for the waveforms with those names). They all take as their main parameter a one-dimensional `ndarray` representing the times at which the signal is to be evaluated. Other parameters control the design of the signal, according to frequency or time constraints.

```
from scipy.signal import chirp, sawtooth, square, gausspulse
import matplotlib.pyplot as plt
t=numpy.linspace(-1,1,1000)
plt.subplot(221); plt.ylim([-2,2])
```

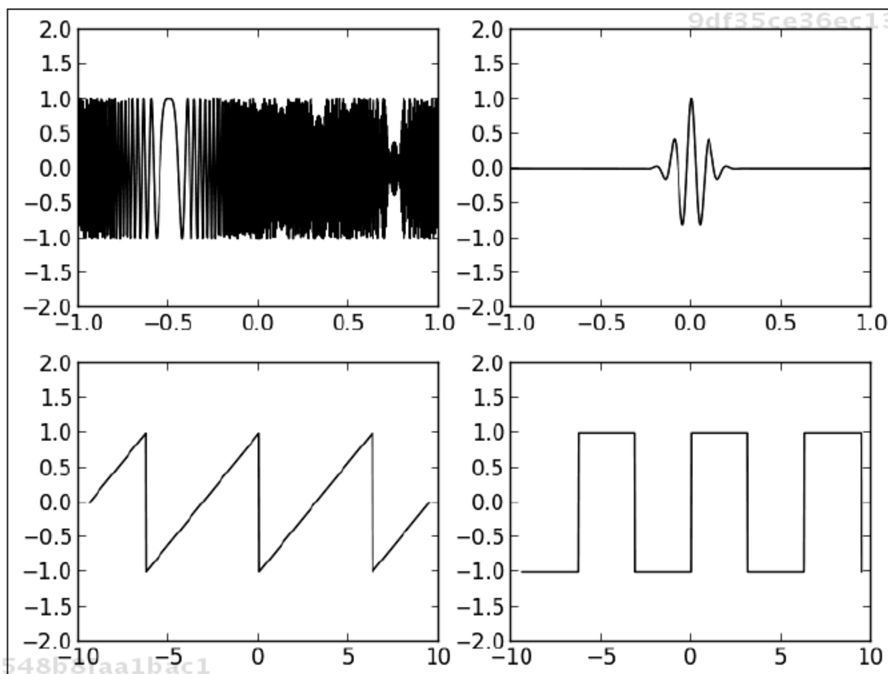

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 94
http://site.ebrary.com/id/10672339?ppg=94
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary


```
plt.plot(t,chirp(t,f0=100,t1=0.5,f1=200))   # plot a chirp
plt.subplot(222); plt.ylim([-2,2])
plt.plot(t,gausspulse(t,fc=10,bw=0.5))      # Gauss pulse
plt.subplot(223); plt.ylim([-2,2])
t*=3*numpy.pi
plt.plot(t,sawtooth(t))                     # sawtooth
plt.subplot(224); plt.ylim([-2,2])
plt.plot(t,square(t))                       # Square wave
```

The following diagram shows waveforms for `chirp` (upper-left), `gausspulse` (upper-right), `sawtooth` (lower-left), and `square` (lower-right):



9df35ce36ec13429fe1548b8faa1bac1
ebrary

The usual method of creating signals is to import them from file. This is possible by using purely NumPy routines, for example `fromfile`:

```
fromfile(file, dtype=float, count=-1, sep='')
```

The `file` argument may point to either a file or a string, the `count` argument is used to determine the number of items to read, and `sep` indicates what constitutes a separator in the original file/string. For images, we have the versatile routine, `imreadin` in either the `scipy.ndimage` or `scipy.misc` module:

```
imread(fname, flatten=False)
```


9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 95
http://site.ebrary.com/id/10672339?ppg=95
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

The `fname` argument is a string containing the location of an image. The routine infers the type of file, and reads the data into array accordingly. In case if the `flatten` argument is turned to `True`, the image is converted to gray scale. Note that, in order to work, the **Python Imaging Library** (**PIL**) needs to be installed.

It is also possible to load `.wav` files for analysis, with the `read` and `write` routines from the `wavfile` submodule in the `scipy.io` module. For instance, given any audio file with this format, say `audio.wav`, the command, `>>>rate,data = scipy.io.wavfile.read("audio.wav")` assigns an integer value to the `rate` variable, indicating the sample rate of the file (in samples per second), and a NumPy `ndarray` to the `data` variable, containing the numerical values assigned to the different notes. If we wish to write some one-dimensional ndarray `data` into an audio file of this kind, with the sample rate given by the `rate` variable, we may do so by issuing the following command:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
>>>scipy.io.wavfile.write("filename.wav",rate,data)
```

Filters

A filter is an operation on signals that either removes features or extracts some component. SciPy has a very complete set of known filters, as well as the tools to allow construction of new ones. The complete list of filters in SciPy is long, and we encourage the reader to explore the help documents of the `scipy.signal` and `scipy.ndimage` modules for the complete picture. We will introduce in these pages, as an exposition, some of the most used filters in the treatment of audio or image processing.

We start by creating a signal worth filtering:

```
from numpy import sin, cos, pi, linspace
f=lambda t: cos(pi*t) + 0.2*sin(5*pi*t+0.1) + 0.2*sin(30*pi*t) +
        0.1*sin(32*pi*t+0.1) + 0.1*sin(47* pi*t+0.8)
t=linspace(0,4,400); signal=f(t)
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We test first the classical smoothing filter of Wiener and Kolmogorov, `wiener`. We present in a plot the original signal (in black) and the corresponding filtered data, with a choice of Wiener window of size 55 samples (in blue). Next we compare the result of applying the median filter, `medfilt` with a kernel of the same size as before (in red):

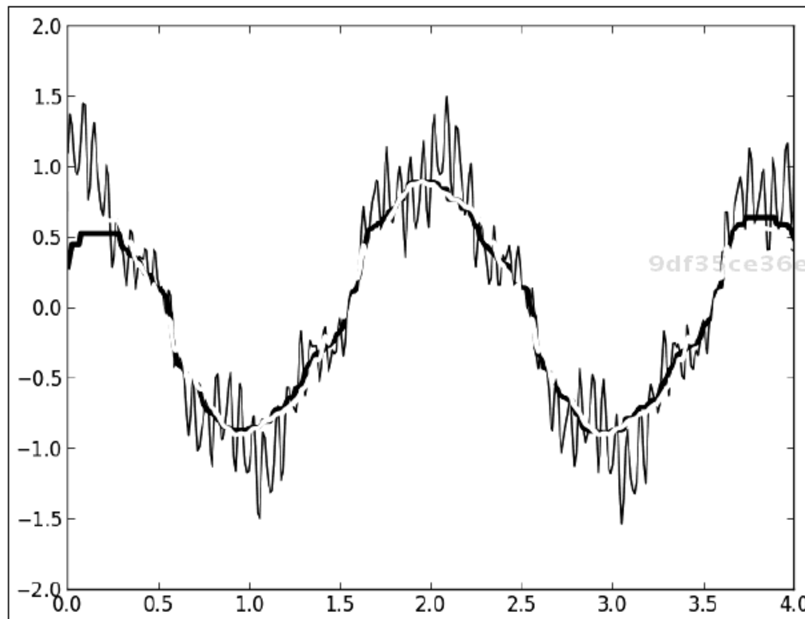
```
from scipy.signal import wiener, medfilt
plt.plot(t,signal,'k')
plt.plot(t,wiener(signal,mysize=55),'b',linewidth=3)
plt.plot(t,medfilt(signal,kernel_size=55),'r',linewidth=3)
```


9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 96
http://site.ebrary.com/id/10672339?ppg=96
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

This gives us the following graph showing the comparison of smoothing filters (`wiener` is the one that has its starting point just below 0.5 and `medfilt` has its starting point just above 0.5):



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Most of the filters in the `scipy.signal` module can be adapted to work in arrays of any dimension. But in the particular case of images, we prefer to use the implementations in the `scipy.ndimage` module, since they are coded with these objects in mind. For instance, to perform a median filter on an image for smoothing, we use `scipy.ndimage.median_filter`. Let us show an example. We will start by loading Lena to array, and corrupting the image with Gaussian noise (zero mean and standard deviation of 16):

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
from scipy.stats import norm     # Gaussian distribution
lena=scipy.misc.lena().astype(float)
lena+=norm(loc=0,scale=16).rvs(lena.shape)
denoised_lena = scipy.ndimage.median_filter(lena)
```

The set of filters for images come in two flavors – statistical and morphological. For example, among the filters of statistical nature, we have the Sobel algorithm oriented to detection of edges (singularities along curves). Its syntax is as follows:

```
sobel(image, axis=-1, output=None, mode='reflect', cval=0.0)
```


9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 97
http://site.ebrary.com/id/10672339?ppg=97
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

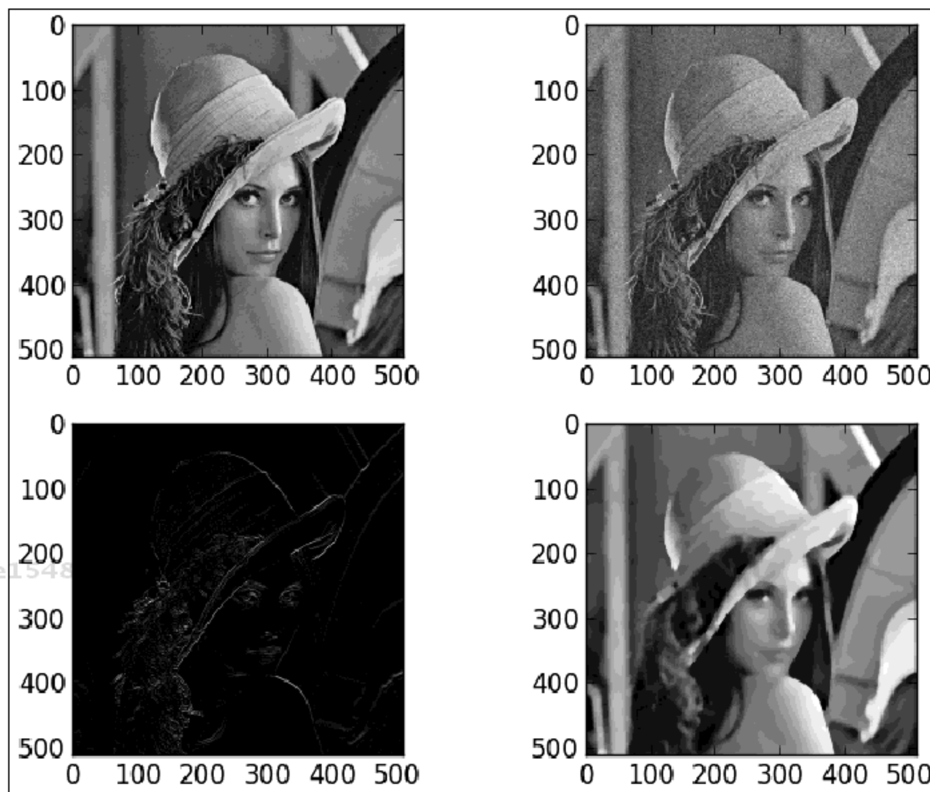
9df35ce36ec13429fe1548b8faa1bac1
ebrary

The optional parameter, `axis` indicates the dimension in which the computations are performed. By default, this is always the last axis (-1). The `mode` parameter, which is one of the strings `'reflect'`, `'constant'`, `'nearest'`, `'mirror'`, or `'wrap'`, indicates how to handle the border of the image, in case there is insufficient data to perform the computations there. In case if `mode` is `'constant'`, we may indicate the value to use in the border, with the `cval` parameter.

```
lena=scipy.misc.lena()
sblX=sobel(lena,axis=0); sblY=sobel(lena,axis=1)
sbl=numpy.hypot(sblX,sblY)
```

The following screenshot illustrates the previous two filters in action—Lena (upper-left), noisy Lena (upper-right), edge map with sobel (lower-left), and median filter (lower-right):

9df35ce36ec13429fe1548b8faa1bac1
ebrary



9df35ce36ec13429fe1548
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 98
http://site.ebrary.com/id/10672339?ppg=98
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

LTI system theory

To investigate the response of a time-invariant linear system to input signals, we have many resources in the `scipy.signal` module. As a matter of fact, to simplify representation of objects, we have a `lti` class (linear-time invariant class) with associated methods such as `bode` (to calculate bode magnitude and phase data), `impulse`, `output`, and `step`.

No matter whether we are working with continuous or discrete-time linear systems, we have routines to simulate such systems (`lsim` and `lsim2` for continuous, `dsim` for discrete), as well as compute impulses (`impulse` and `impulse2` for continuous, `dimpulse` for discrete) and steps (`step` and `step2` for continuous, `dstep` for discrete).

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Transforming a system from continuous to discrete is possible with `cont2discrete`, but in either case we are able to provide for any system with any of its representations, as well to convert from one to another. For instance, if we have the zeros `z`, poles `p`, and system gain `k` of the transfer function, we may obtain the polynomial representation (numerator first, then denominator) with `zpk2tf(z,p,k)`. If we have numerator (`num`) and denominator (`dem`) of the transfer function, we obtain the state-space with `tf2ss(num,dem)`. This operation is reversible, with the `ss2tf` routine. The change of representation from zero-pole-gain to/from state-space is also contemplated in the (`zpk2ss`, `ss2zpk`) module.

Filter design

There are routines in the `scipy.signal` module that allow the creation of different kinds of filters with diverse methods. For instance, the `bilinear` routine returns a digital filter from an analog using a bilinear transform. **Finite impulse response** (**FIR** for short) filters can be designed by the window method with the `firwin` and `firwin2` routines. **Infinite impulse response** (**IIR** for short) filters can be designed in two different ways, via `iirdesign` or `iirfilter`. Butterworth filters can be designed with the `butter` routine. There are also routines to design filters of Chebyshev (`cheby1`, `cheby2`), Cauer (`ellip`), and Bessel (`bessel`).

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Window functions

And no signal processing computational system would be complete without an extensive list of windows—mathematical functions that are zero valued outside specific domains. In this section, we will use a few of the windows coded in the `scipy.signal` module to design very simple smoothing filters by convolution. We will be testing them on the same one-dimensional signal we employed before, for comparison.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 99
http://site.ebrary.com/id/10672339?ppg=99
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We will start by showing the plot of four well-known window functions – Boxcar, Hamming, Blackman-Harris (Nuttall version), and triangular. We will use a size of 31 samples:

```
from scipy.signal import boxcar, hamming, nuttall, triang
windows=['boxcar', 'hamming', 'nuttall', 'triang']
for w in windows:
eval( 'plt.plot(' + w + '(31))' )
plt.ylim([-0.5,2]); plt.xlim([-1,32])
plt.legend(windows)
```

We need to extend the original signal by fifteen samples for plotting purposes:

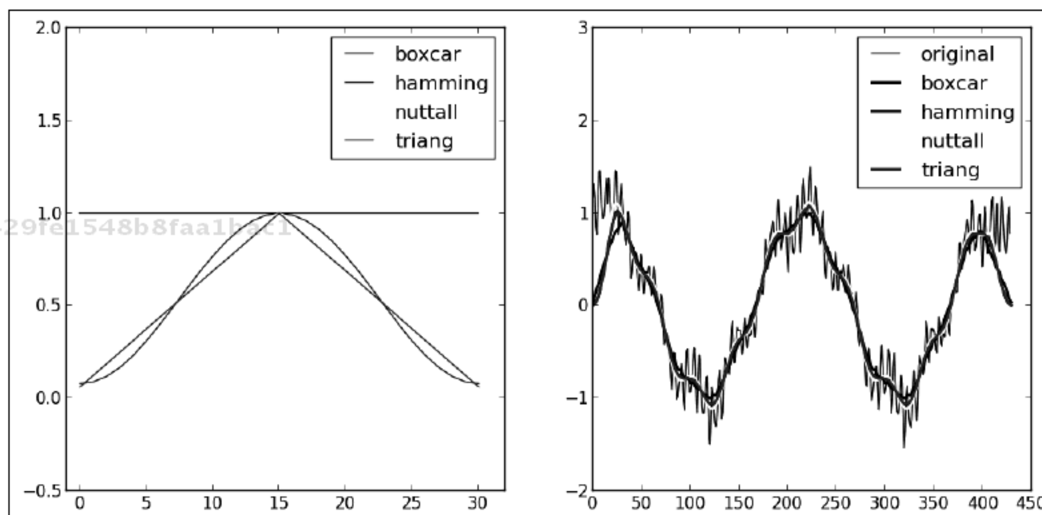
9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
extended_signal=numpy.r_[signal[15:0:-1],signal,signal[-1:-15:-1]]
plt.plot(extended_signal,'k')
```

The final step is the filter itself, which we perform by a simple convolution:

```
for w in windows:
    window = eval( w+'(31)')
    output=numpy.convolve(window/window.sum(),signal)
plt.plot(output,linewidth=2)
plt.ylim([-2,3]); plt.legend(['original'+windows)
```

This produces the following output showing convolution of a signal with different windows:



9df35ce36ec13429fe1548b8faa1bac1
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 100
http://site.ebrary.com/id/10672339?ppg=100
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Image interpolation

The set of filters on images that perform some geometric manipulation of the input is classically termed image interpolation, since this numerical technique is the root of all the algorithms. As a matter of fact, SciPy collects all these under the `submodulescipy.ndimage.interpolation` for ease of access. This section is best explained through examples, going over the most meaningful routines for geometric transformation. The starting point is the image Lena. We now assume that all functions from the submodule have been imported into the session.

We need to apply an affine transformation on the domain of the image, given in matrix form as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

$$L(x,y) = \underbrace{\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}}_{A} \begin{pmatrix} x \\ y \end{pmatrix} + \underbrace{\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}}_{b}$$

To apply the transformation on the domain of the image we will issue the `affine_transform` command (note the syntax is self explanatory):

```
A=numpy.mat("0,1;-1,1.25"); b=[-400,0]
Ab_Lena=affine_transform(lena,A,b,output_shape=(512*2.2,512*2.2))
```

For a general transformation, we use the `geometric_transform` routine with the following syntax:

```
geometric_transform(input, mapping, output_shape=None,
                    output=None, order=3, mode='constant',
cval=0.0, prefilter=True, extra_arguments=(),
extra_keywords={})
```

We need to provide a rank-2 map from tuples to tuples as the parameter mapping. For instance, we desired to apply the Möbius transform for complex-valued number `z` (where we assume the values of `a`, `b`, `c`, and `d` are already defined and they are complex-valued numbers).

9df35ce36ec13429fe1548b8faa1bac1
ebrary

$$f(z) = \frac{az + b}{cz + d}$$

We would have to code it in the following way:

```
def f(z):
    temp = a*(z[0]+1j*z[1]) + b
    temp /= c*(z[0]+1j*z[1])+d
    return (temp.real, temp.imag)
```


9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 101
http://site.ebrary.com/id/10672339?ppg=101
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

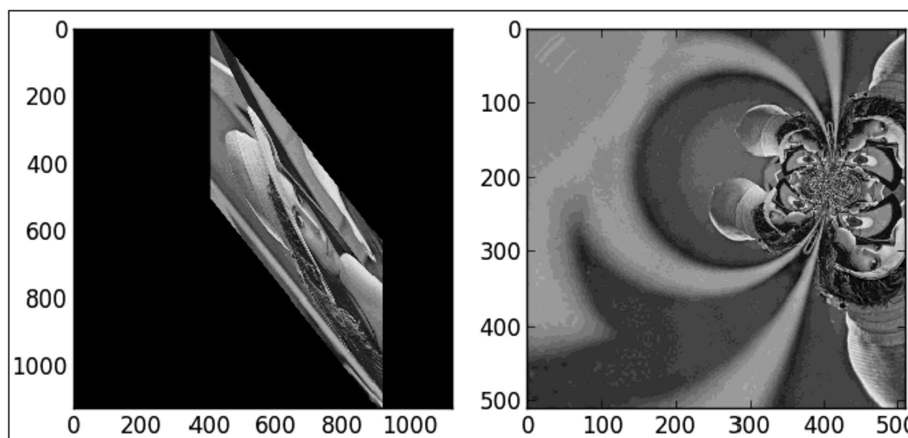
In both functions, the values of the grid that cannot be computed directly with the formula are inferred with spline interpolation. We may specify the order of this interpolation with the `order` parameter. The points outside the domain of definition are not interpolated, but filled according to some predetermined rule. We may impose this rule by passing a string to the `mode` option. The choices are – `'constant'`, to use a constant value that we may impose with the `cval` option; `'nearest'`, that continues the last value of the interpolation on each level line; `'reflect'` or `'wrap'`, which are self explanatory.

For example, for the values `a = 2**15*(1+1j),b = 0,c = -2**8*(1-1j*2)`, and `d = 2**18-1j*2**14`, we obtain (after imposing the `reflect` mode) the result, as shown just after this line of code:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
Moebius_Lena = geometric_transform(lena,f,mode='reflect')
```

The following screenshot shows affine transformation (left) and geometric transformation (right):



9df35ce36ec13429fe1548b8faa1bac1
ebrary

For the special cases of rotations, shifts, or dilations, we have the syntactic sugar routines `rotate(input,angle)`, `shift(input, offset)`, and `zoom(input,dilation_factor)`.

Given any image, we know the value of the array at pixel values (with integer coordinates) in the domain. But, what would be the corresponding value of a location without integer coordinates? We may obtain that information with the valuable routine, `map_coordinates`. Note that the syntax may be confusing, especially with the parameter coordinates:

```
map_coordinates(input, coordinates, output=None, order=3,
                mode='constant', cval=0.0, prefilter=True)
```


9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 102
http://site.ebrary.com/id/10672339?ppg=102
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

For instance, if we wish to evaluate Lena at the locations (10.5, 11.7) and (12.3, 1.4), we collect the coordinates as a sequence of sequences; the first internal sequence contains the x values, and the second, the y values. We may specify the order of splines used with order, and the interpolation scheme outside of the domain, if needed, as in the previous examples.

```
>>>lena=scipy.misc.lena().astype(float)
>>> coordinates=[[10.5, 12.3], [11.7, 1.4]]
>>>map_coordinates(lena, coordinates, order=1)
array([ 157.2 ,  157.42])
>>>map_coordinates(lena, coordinates, order=2)
array([ 157.80641507,  157.6741489 ])
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Morphology

We also have the possibility of creating and applying filters to images based on mathematical morphology, both to binary and gray-scale images. The four basic morphological operations are opening (binary_opening), closing (binary_closing), dilation (binary_dilation), and erosion (binary_erosion). Note that the syntax for each of these filters is very simple, since we only need two ingredients – the signal to filter and the structuring element to perform the morphological operation.

```
binary_operation(signal, structuring_element)
```

We have illustrated the use some of these operations towards an application to obtain the structural model of an oxide, but we postpone this example until we cover the notions of triangulations and Voronoi diagrams in *Chapter 7*, *SciPy for Computational Geometry*.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We may use combinations of these four basic morphological operations to create more complex filters for removal of holes, hit-or-miss transforms (to find the location of specific patterns in binary images), denoising, edge detection, and many more. The module even provides with some of the most common filters constructed this way. For instance, for the location of the letter "e" in a text (which we covered previously as an application of correlation), we could use the following command instead:

```
>>>binary_hit_or_miss(text, letterE)
```

For gray-scale images, we may use a structuring element or a footprint. The syntax is, therefore, a little different:

```
grey_operation(signal, [structuring_element, footprint, size, ...])
```


9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 103
http://site.ebrary.com/id/10672339?ppg=103
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

If we desire to use a completely flat and rectangular structuring element (all ones), then it is enough to indicate the size as a tuple. For instance, to perform gray-scale dilation of a flat element of size (15,15) on our classical image of Lena, we issue the following command:

```
>>>grey_dilation(lena, size=(15,15))
```

The last kind of morphological operations coded in the scipy.ndimage module perform distance and feature transforms. Distance transforms create a map that assigns to each pixel the distance to the nearest object. Feature transforms provide with the index of the closest background element instead. These operations are used to decompose images into different labels. We may even choose different metrics such as Euclidean distance, chessboard distance, and taxicab distance. The syntax for the distance transform using a brute force algorithm is as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
distance_transform_bf(signal, metric='euclidean', sampling=None,
return_distances=True, return_indices=False,
                        distances=None, indices=None)
```

We indicate the metric with the strings such as 'euclidean', 'taxicab', or 'chessboard'. If we desire to provide the feature transform instead, we switch return_distances to False and return_indices to True.

Similar routines are available with more sophisticated algorithms – distance_transform_cdt (using chamfering for taxicab and chessboard distances). For Euclidean distance, we also have distance_transform_edt. All these use the same syntax.

Summary

In this chapter we explored signal processing (any dimensional) including the treatment of signals in frequency space, by means of their Discrete Fourier Transforms. These correspond to the fftpack, signal, and ndimage modules.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 104
http://site.ebrary.com/id/10672339?ppg=104
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 105
http://site.ebrary.com/id/10672339?ppg=105
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.