Lab 9: FTT and power spectra

The Fast Fourier Transform (FFT) is a fast and efficient numerical algorithm that computes the Fourier transform. The *power spectrum* is a plot of the power, or variance, of a time series as a function of the frequency[1]. If $G(f)$ is the Fourier transform, then the power spectrum, $W(f)$, can be computed as

$$W(f) = |G(f)| = G(f)G^*(f)$$

where $G^*(f)$ is the complex conjugate of $G(f)$. We refer to the power spectrum calculated in this way as the *periodogram*.

Currently, many investigators prefer to estimate the *power spectral density* using `matplotlib.mlab.psd()`. This method is based on Welch's averaged periodogram method. Welch's method reduces noise in the estimated spectrum at the expense of reducing the frequency resolution (see below). For many of the experimental uses of power spectra, the advantages of reducing the effects of noise out-weigh the dis-advantages of reduced frequency resolution. As we showed in lecture there is little practiacl difference between determining the periodogram versus the power spectral density. However, the flexibility of `mlab.psd()` provides esufficient motivation to learn how to use this package.

Often overlooked is that the fact that $W(f)$ is the power spectrum obtained for an infinitely long time series measured with infinitely fine precision. In contrast, in the laboratory we work with time series of finite length that are subjected to uncorrelated, random inputs (noise) and effects introduced by the process of measuring the signal.

The purpose of this laboratory is to illustrate the use of `mlab.psd()` and explore a number of applications of the FFT and power spectra.

[1]For simplicity we have assumed that the independent variable is time; however, it could also be a spatial dimension. We use the term *power spectra* as a collective term to include both the periodogram and the power spectral density.

Browser: Essentially everything that you would ever want to know about the uses of the FFT can be located on the Internet by asking the right questions. In Python, the functions necessary to calculate the FFT are located in the `numpy` library called `fft`. If we want to use the function `fft()`, we must add the following command to the top matter of our program:

```
import numpy.fft as fft
```

Thus, the command for determining the FFT of a signal $x(t)$ becomes `fft.fft(x)`. Of course, you could import the `fft`-package from `numpy` under a different name; however, this might make the program less readable by others. Other functions related to the use of the FFT are located in `scipy` such as the library `signal`, i.e.

```
import scipy.signal as signal
```

The description of each command and how to use it can easily be obtained by using your web browser and typing the name of the library together with the function that you want to know about.

House keeping: Today we will illustrate the applications of the FFT and the power spectra by working with real data. The deliverables take the form of figures prepared using `Matplotlib`. We suggest that you create a directory for each figure which contains both the data and the program(s) used to generate the required figure. This is actually a good habit to acquire. One advantage is that you don't need to worry about specifying the required paths since, by default, a Python program looks first in the directory they are running for required input files. A second advantage occurs if, my goodness, you actually need to modify your figure at a later date following comments made by a grader or a reviewer of your article (senior thesis, publications, book).

j represents the imaginary number, which makes sense since we are using oscillating waves.

1 Background

We illustrate the difference between $W(f)$ and $w(f)$ by calculating the power spectrum of $x(t) = \sin(2\pi f_0 t)$, where f_0 is a particular frequency. By definition, the Fourier transform of $\sin(2\pi f t)$ is

$$G(\sin 2\pi f_0 t)(f) = \int_{-\infty}^{\infty} e^{-2\pi j f t} \sin 2\pi f_0 t dt \qquad (1)$$

The definition of the continuous Fourier transform.

where we have distinguished a particular value of the frequency, f, as f_0. Using Euler's relation, we can write

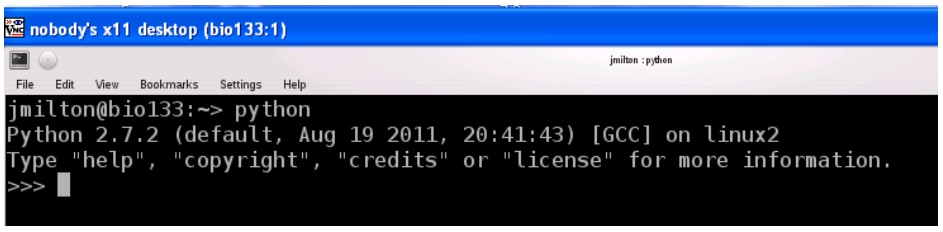
$$\sin 2\pi f_0 t = \frac{e^{2\pi j f_0 t} - e^{-2\pi j f_0 t}}{2j}$$

Substituting into (1), we obtain

$$\begin{aligned} G(\sin 2\pi f_0 t)(f) &= \int_{-\infty}^{\infty} e^{-2\pi j f t} \left[\frac{e^{2\pi j f_0 t} - e^{-2\pi j f_0 t}}{2j} \right] dt \\ &= \frac{1}{2j} \int_{-\infty}^{\infty} \left[e^{-2\pi j (f-f_0)t} - e^{-2\pi j (f+f_0)t} \right] dt \\ &= \frac{1}{2j} \left[\delta(f-f_0) - \delta(f+f_0) \right] \\ &= \frac{j}{2} \left[\delta(f+f_0) - \delta(f-f_0) \right] \end{aligned}$$

Use delta to represent the exponent integral. That is the form of the delta function.

Here we see that both the Fourier transform and the power spectrum, $W(f)$, of $\sin(2\pi f t)$ are predicted to be composed of two delta-functions, one centered at $+f_0$ and the other at $-f_0$.

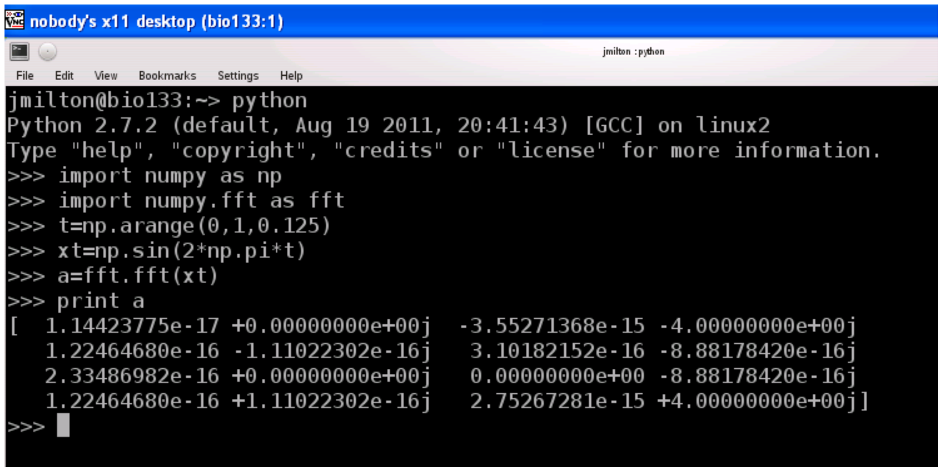

nobody's x11 desktop (bio133:1)
jmilton : python
File Edit View Bookmarks Settings Help

```
jmilton@bio133:~> python
Python 2.7.2 (default, Aug 19 2011, 20:41:43) [GCC] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Figure 1: Computer screen shot obtained after typing **python** at the Command prompt. The three $>>>$ means that we are operating in script mode.

Now let's use Python to compute the FFT and the power spectrum, $w(f)$. Python can be run directly from the command line, namely in an interactive mode[2] (a much more powerful and popular version of interactive Python programming

[2]Up to now we have run Python in its script mode, namely, we write computer program, `name.py` and then run the program by typing `python name.py` (Did you remember to add the & ?). There are two advantages: 1) we can use the same program over and over again, and 2) the program gives us a permanent record of what we did. However, the interactive mode is useful when the goal is simply to explore data.



nobody's x11 desktop (bio133:1)
jmilton :python
File Edit View Bookmarks Settings Help

```
jmilton@bio133:~> python
Python 2.7.2 (default, Aug 19 2011, 20:41:43) [GCC] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> import numpy.fft as fft
>>> t=np.arange(0,1,0.125)
>>> xt=np.sin(2*np.pi*t)
>>> a=fft.fft(xt)
>>> print a
[  1.14423775e-17 +0.00000000e+00j  -3.55271368e-15 -4.00000000e+00j
   1.22464680e-16 -1.11022302e-16j   3.10182152e-16 -8.88178420e-16j
   2.33486982e-16 +0.00000000e+00j   0.00000000e+00 -8.88178420e-16j
   1.22464680e-16 +1.11022302e-16j   2.75267281e-15 +4.00000000e+00j]
>>>
```

Figure 2: Obtaining the fft of a 1s 4Hz sine wave by running Python in script mode. Note that most of the numbers for the fft are very small: there are two exceptions. See text for discussion.

is `iPython`). To enable the interactive mode type `python` in your command window (Figure 1). Type the lines of Python code shown in Figure 2 to obtain the FFT of a 1 Hz sine wave.

What does the output on the screen mean? First we note that there are 8 numbers (the $\sin(2\pi f_0 t)$ was digitized to give 8 data points per second), all of the numbers are written in the form of a complex number, and Python uses the convention that $j = \sqrt{-1}$. By convention the FFT is outputted using *reverse-wrap-around ordering*. For this example, the ordering of the frequencies is

$$[0, 1, 2, 3, 4, -3, -2, -1]$$

Note that the 8 discrete data points yield 8 Fourier coefficients and that the highest frequency that will be resolved is the Nyquist frequency, $N/2$. The first half of the list of numbers corresponds to the positive frequencies and the second half to the negative frequencies. We suspect that most of you expected to see on the computer screen

```
0.0000
-0.0000     -4.0000j
0.0000       0.0000j
0.0000       0.0000j
0.0000
```

```
 0.0000      0.0000j
 0.0000      0.0000j
-0.0000     +4.0000j
```

However, notice that many of the numbers are very, very small, for example of the order 10^{-17}. These numbers simply represent numerical noise in the computer. If we set mentally all numbers $< 10^{-3}$ equal to 0, then we obtain the expected result.

It should be noted that recent releases of `numpy.fft` include the commands `rfft()` and `irfft()`. The advantage of using `rfft()` over `fft()` becomes of practical importance when we consider a problem that can arise when we compute the inverse FFT using `ifft()`. Since we typically have a real signal, we expect that the inverse should also be real. However, it can happen, due to rounding errors, that `ifft()` contains complex numbers. The use of `rfft()` avoids this problem.

Thanks to the development of computer software packages, such as MATLAB, Mathematica, Octave and Python, this task has become much easier. A particularly useful function is Python's `matplotlob.mlab.psd()` developed by the late John D. Hunter[3]

```
matplotlib.mlab.psd(x, NFFT, Fs, detrend, window,
noverlap=0, pad_to, sides=, scale_by_freq)
```

It is important to keep in mind that `psd()` does not calculate the periodogram, but calculates the power spectral density using a mathematical method known as Welch's method. The advantages of using `mlab.psd()` are that it is very versatile and for everyday use we can accept the default choices for most of the options.

2 Exercise 1: Computing the power spectral density using mlab.psd()

2.1 Power Spectral Density Recipe

In the not too distant past, obtaining proper power spectra was a task that was beyond the capacity of most biologists[4]. However, thanks to the development

[3]We strongly recommend that readers do not use pylab's version of psd(). The matplolib.mlab.psd() produces the same power spectrum as obtained using the corresponding programs in MATLAB.

[4]For convenience we reproduce Section 8.4.5 here.

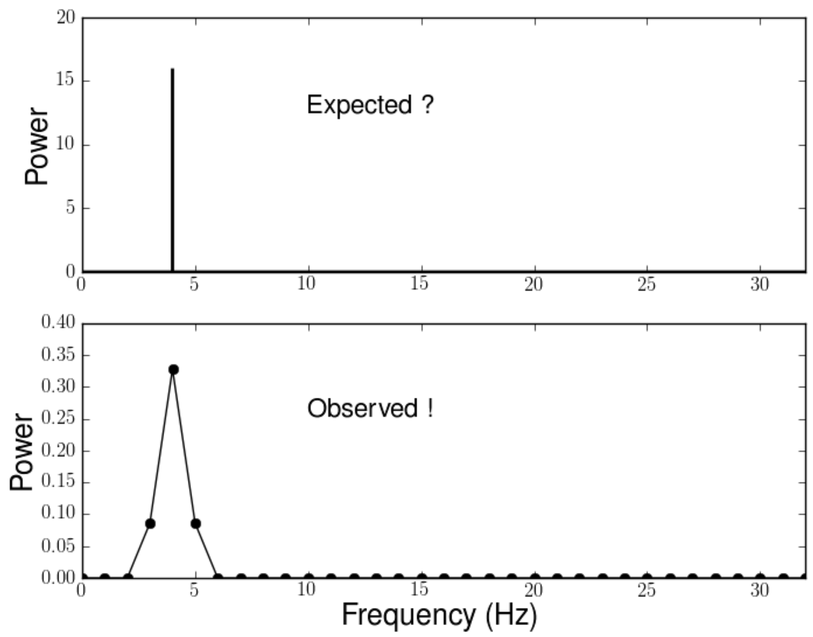


Figure 3: a) One-sided periodogram computed for 1s of a 4Hz sine wave using (1). b) One-sided power spectral density computed using mlab.psd() for the same signal used in a).

of computer software packages such as MATLAB, Mathematica, Octave, and Python, this task has become much easier. Although it is relatively easy to write a computer program that computes the periodogram, it is not so easy to write the program in a manner that is convenient for many of the scenarios encountered by an experimentalist. For this reason we strongly recommend the use of Python's `matplotlob.mlab.psd()`, developed by the late John D. Hunter using the method described by Bendat and Piersol [1].[5] This function has the form

```
power, freqs = matplotlib.mlab.psd(x, NFFT, Fs,
detrend, window, noverlap=0, pad_to, sides=,
scale_by_freq)
```

[5]Do not use PyLab's version of `psd()`. The function `matplolib.mlab.psd()` produces the same power spectrum as that obtained using the corresponding programs in MATLAB.

The output is two one-dimensional arrays, one of which gives the values of the power in db Hz^{-1}, the other the corresponding values of the frequency.

The advantage of `matplotlib.mlab.psd()` is its versatility. Assuming that an investigator has available a properly low-pass-filtered time signal, the steps for obtaining the power density density are as follows:

1. Enter the filtered and discretely sampled time signal $x(t)$.

2. The number of points per data block is `NFFT`. NFFT must be an even number. The computation is most efficient if $\texttt{NFFT} = 2^n$, where n a positive integer; however, with modern laptops, the calculations when $\texttt{NFFT} \neq 2^n$ are performed quickly.

3. Enter `Fs`. This parameter scales the frequency axis on the interval $[0, Fs/2]$, where $Fs/2$ is the Nyquist frequency.

4. Enter the detrend option. Recall that it this a standard procedure to remove the mean from the time series before computing the power spectrum.

5. Enter the windowing option. Keep in mind that using a windowing function is better than not using one, and from a practical point of view, there is little difference among the various windowing functions. The default is the Hanning windowing function.

6. Typically accept the `overlap` default choice of 0 (see below for more details).

7. Choose `pad_to` (see below). The default is 0.

8. Choose the number of sides. The default for real data is a one-sided power spectrum.

9. Choose `scale_by_freq`. Typically, you will choose the default, since that makes the power spectrum compatible with that obtained using MATLAB.

The versatility of `mlab.psd()` resides in the interplay between the options *NFFT*, `noverlap` and `pad_to`. The following scenarios illustrate the salient points for signals for which the mean has been removed.

1. Frequencies greater than 1 Hz: The options available depend on the length of x, $L(x)$. If $L(x) =$ NFFT, then the time is 1 sec and we can accept the defaults and the command to produce the power spectrum. For example, when the digitization rate is 256 points per second the command would be

```
mlab.psd(x,256,256)
```

We can increase the resolution by using the `pad_to` option. For example for a sample initially digitzied at 256 Hz, we can calculate the power spectrum for a 512 Hz digitization rate by using the command

```
mlab.psd(x,256,512,pad_to=512)
```

Finally we can average the time series by taking x to be an integer mutiple of `NFFT`, X_n. This prodedure is useful when data is noisy and when we want to use the `noverlap` option in addition to windowing to minimize power leakage. Thus the command takes the form

```
mlab.psd(X_n,256,256,noverlap=128)
```

It should be noted that the optimal choice for `noverlap` is one-half of `NFFT`, namely the length of the time series, including zero padding.

2. Frequencies less than 1 Hz: It is necessary to first determine the minimal length of time series that is required. Suppose we wanted to look for periodic components of the order of 0.1Hz. A $0.1Hz$ sinusoidal rhythm will have 1 cycle every 10 seconds. Thus the length of time series must be at least 10 seconds long. However, we will clearly get a much better looking power spectrum the longer the time series so maybe we should use 10 times longer, e.g. 100s. Thus the command becomes

```
mlab.psd(x,6400,64)
```

It is important to note that the characterization of the frequency content of biological time series in the less than 1 Hz freqnecy range is made problematic because of the effects of non-stationarity. Thus the length of the time series and often the particular segment of the time series to be analyzed must be chosen with great care. In fact, it is in the analysis of such time series that many of the options available in `mlab.psd()` are most useful.

2.2 Example using mlab.psd()

Figure 3 shows the power spectrum determined for 1s of a 4 Hz sine wave sampled at 64 Hz. Note that we have plotted the results using a linear scale for both axis. What is the Nyquist frequency? It is $64/2 = 32$ Hz. The periodogram is shown in Figure 3a. We obtained the periodogram by squaring (term-by-term) the `fft` shown in Figure 2. Since values except those at ± 4Hz are zero we get two delta-functions, each with with height 16. We have plotted the one-sided power spectrum.

The power spectral density is shown in Figure 3b. The power spectral density looks different than the periodogram. Although both are centered at the expected 4Hz, the power spectral density is broader than the periodogram. A rough estimate of the frequency resolution of a pure frequency in the power spectrum is the width of the peak at one-half height: the bigger the half width the lower the frequency resolution. Thus the increased width of the 4Hz peak in Figure 3b reflects the decrease in frequency resolution obtained when on uses Welch's method to estimate the power spectral density. Keep in mind that the advantage of Welch's method occurs when we have imperfect data of finite length subjected to random perturbations.

Figure 4 shows the loglog plots of the power spectra shown in Figure 3. We suspect that most of you are surprised by the appearance of loglog plot of the power spectral density. Why does it look this way? To answer this question it is important to remember that the power spectrum is actually being computed for the signal that has been filtered using the rectangular sampling function. Thus the power spectrum of a discrete frequency will resemble `sinc(f)`. The reason that the ripples of the `sinc` function are not apparent is because of the averaging (smoothing) that is part of Welch's method to determine the power spectral density.

Questions to be answered:

1. Frequency resolution: The frequency resolution of the power spectral density can be increased by increasing the length of the time series. In `mlab.psd()` we can increase the sample length by increasing the number of points in the data block, i.e. by changing `NFFT`, but by keeping F_s the same. For example, if we had 10s of a 4Hz sine wave sampled at 64Hz, then $NFFT = 640$ and $F_s = 64$. An alternate strategy is to pad with zeros. Modify `sine_plot.py` to test these two methods, in each case observing what happens to the width of the peak at half-height.

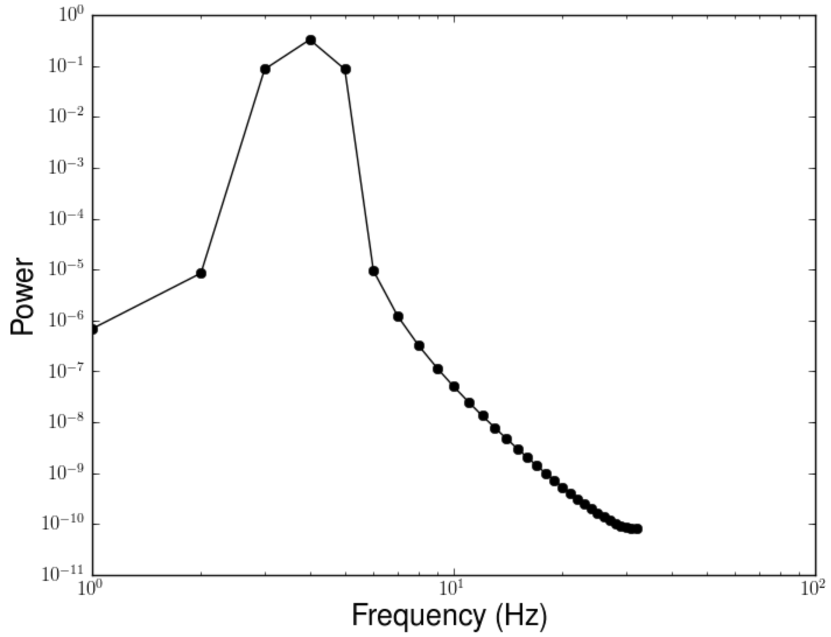


Figure 4: Loglog plot of power spectral density determined using mlab.psd() for 1s of a 4Hz sine wave. Note that we used the endpoint=False option in np.linspace(). What happens to the power spectral density function when we choose the Default option, i.e. endpoint=True? Why?

2. **Aliasing:** What happens if we add to xt a second sine term having a frequency greater than 32Hz? Write a program in which you determine the power spectrum for the signal

$$xt(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$$

sampled at 64 Hz. Take, for example, $f_1 = 4$ and $f_2 = 50$. You will see two peaks in the power spectrum; one at 4Hz and the other at second frequency less than the Nyquist frequency, f_{nq}. The appearance of a frequency component higher than f_{nq} as a frequency lower than f_{nq} is referred to as *aliasing*. In other words, when frequencies higher than the Nyquist frequency, f_0, are present, then these higher frequencies will be falsely reproduced as lower frequencies. It is this effect that is called aliasing. You can predict the aliased position of the high frequency higher than the Nyquist frequency by using the relationship

$$\sin(2\pi[2f_{Nyq} - f]t_n + \phi) = -\sin(2\pi f' t_n - \phi) ,$$

where ϕ is the phase and $t' = 2f_{\text{Nyq}}f$. Thus for example, for a compact disk, the Nyquist frequency is $f_{\text{nq}} = 22050$. Hence a signal at a frequency of $34,100$ Hz will look exactly like a signal at a frequency of $44,100 - 34,100 = 10,000$Hz. Use this equation to predict where the aliased sine wave peak will appear in your power spectrum?

3. The data file, `sway.csv`, is a two minute recording of the fluctuations in the center of pressure (COP) for a healthy adult standing quietly on a force platform with eye closed. The data is sampled at 200 Hz. The first column in the data file is the x-coordinate (medio-lateral) of the COP and the second column is the y-coordinate (anterior-posterior). Write a Python program to calculate $X = \sqrt{x^2 + y^2}$ and then determine the power spectral density using `psd()`. Do you think that there is a significant periodic component present?

   The data file, `fish_1s_10000.tsv`, is 1 second of the electrical signals generated by a Keck Science Department electric fish sampled at 10000 Hz. It is two columns of data: the first column is the time and the second column is the voltage. The species of electric fish can often be determined from the power spectrum [4]. A copy of the paper by Fugere and Kruhe on electric fish power spectra will be supplied to you in the lab. Using the results of this paper determine which type of electric fish corresponds to the one you calculated the power spectrum for.

3 Exercise 2: Low-frequency signals

In biology, we are often interested in detecting frequencies that are less than 1 Hz. Examples include the resting adult respiratory rhythm ($0.13 - 0.33$Hz), circadian rhythms ($\approx 10^{-5}$Hz), a 10-year population cycle ($\approx 10^{-9}$). How do we use `mlab.psd()` to calculate the power spectral density of such signals?

The important question to ask is what is the minimal length of time series that is required. Suppose we wanted to look for periodic components of 0.1Hz. A $0.1Hz$ rhythm will have 1 cycle every 10 seconds. Thus the length of time series must be at least 10 seconds long. However, we will clearly get a much better looking power spectrum the longer the time series so maybe we should use 10 times longer, e.g. 100s. In `mlab.psd()` we use the same trick as described above to increase the resolution, namely, we set for example `NFFT=6400` and `Fs=64`. We will return to this problem in Laboratory 11. However, you should verify that this procedure works for calculating the power spectral density of a 0.1 Hz sine wave.

There are a number of important issues related to the characterization of the properties of low frequency signals. The most important are related to the fact that as the frequency we wish to resolve become slower, the length of the time series needed to resolve the signal rapidly becomes very long. As the length of the required signal becomes longer, issues related to the stationarity of the signal become important. For example, there can be slow DC-baseline drifts in the outputs of measurement devices, artifacts related to movement and sweating and as in the case of the EEG, biological systems tend to naturally be continuously varying. Thus it is possible that a low frequency signal of interest can be lost in the non-stationarity of the process. Another problem arises because of the prevalence of noise in biological system which typically contributes power to the spectrum proportional to $1/f$. Consequently low-frequency signals can be lost in the noise.

4 Exercise 3: FFT Applications

4.1 Convolution:

Consider our black box analogy of a dynamical system. If the dynamical system is linear then the output for any input can be determined using the convolution integral as

$$\text{output }(t) = \int_{-\infty}^{\infty} \text{impulse }(t) \text{ input }(t-u) du \ .$$

If we use the Fourier transform then

$$\text{Output }(f) = \text{Impulse }(f) \text{ Input }(f) \ , \tag{2}$$

where the capital letters have been used to indicate the Fourier transform. Since the required Fourier transforms can be performed numerically using the FFT, (2) provides another method for solving the convolution integral.

However, there are practical problems using the FFT to solve the convolution integral. The mathematical description of the integral transforms involves time series that are infinitely long. On the other hand, experimentally attained time series are always finite. Thus the will be effects, collectively referred to as *end effects*, that effect the evaluation of the integral transform and consequently the evaluation of the convolution integral. The practical solution is to pad the time series with zeros [5]!

Performing the convolution integral assumes that 1) the length of the impulse response function is the same as the length of the input time series, and 2) the input time series is periodic. The solution to the first problem is straight-forward: we pad the impulse response function by extending it with 0's so that its total length equals that of the input time series. In order to understand how the second problem can be dealt with it is useful to refer to the graphical interpretation of the convolution integral that we introduced in Laboratory 8. As a consequence of folding, displacement and multiplication, a portion of each end of the input time series is erroneously wrapped around by convolution with the impulse response function. Thus we need to pad one end of the input time series with zeros to ensure that the convolution integral is not contaminated from the effects of this wrap around effect. If the length of the time series is N and of the impulse function is M, then the input time series is padded at one end with $N + M$ zeros [5].

The program used in Laboratory 7 used `np.convolve()` to solve the convolution integral can be modified to use `np.fft.fft()` to do the same exercise. The program is called `convolve_fft.py`. We have modified this program with the lines that pad with zeros deleted. You are asked to supply the missing lines (if you can't figure this out you can check the lines in the programs `convolve_fft_neuron.py` or `convolve_fft_sine.py` on the website).

4.2 Filtering images

Convolution of an input with a linear filter in the temporal domain is equivalent to multiplication of the Fourier transforms for the input and the filter in the frequency domain. This provides a conceptually simple way to think about filtering: transform your signal into the frequency domain, dampen the frequencies that you are not interested in by multiplying the frequency spectrum by the desired weights, and then do an inverse transform to get the filtered signal.

To illustrate consider the picture shown in the top left corner of Figure 5 was taken by a camera attached to a robot on the surface of the moon (`moonlanding.png`). The picture is contaminated with high frequency noise related to the generation and transmission of the image back to earth. How can we filter away the noise so that we can see the sharper image shown in the lower left hand corner of Figure 5?

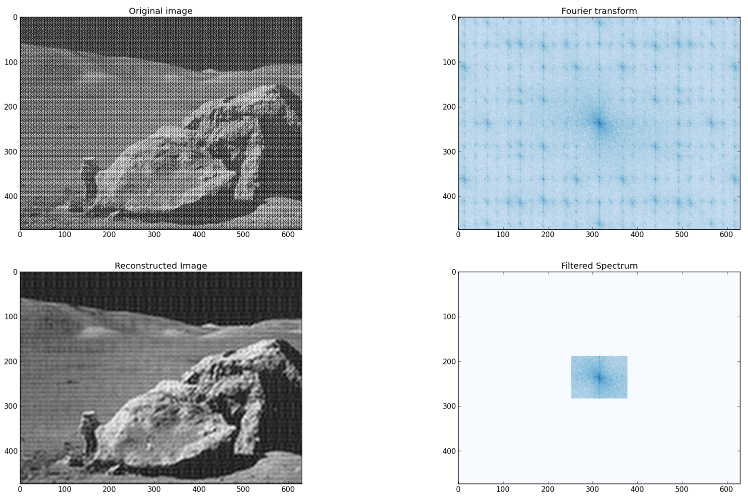


Figure 5: High frequency noise filtering of a 2-D image in the Fourier domain. The upper two panels show the original image (left) and its spectral power (right). The lower two panels show the same data with the high frequency power set to zero.

The power spectrum of a natural scene has the form

$$\text{power (f)} \approx \frac{1}{f^{\alpha}}$$

where $\alpha \approx 2$ [2, 3, 8, 6, 7]. This observation means that the frequency content is predominantly in the low frequency range. By convention the 2-D Fourier transform of an image is plotted such that the center of the image corresponds to low frequencies and the high frequencies are located at the corners of the image. Unfortunately, the `np.fft.fft2()` function in both MATLAB and Python plots the data in exactly the opposite manner, namely high frequency components located in the center and low frequency components at the corners. Fortunately the function `np.fft.fftshift()` re-orders the frequencies correctly. The 2-D Fourier transform of the image shown in the top left corner of Figure 5 is shown in the top right corner.

A simple way to remove the noise is to use a *brick wall* filter. This is accomplished by setting the amplitudes of all of the frequency components greater that a cut-off frequency to zero. The lower right-hand corner of Figure 5 shows the 2-D

Fourier transform of the brick all filtered image. This is not always the best way to do this filtering: sharp edges in one domain usually introduce artifacts in another. However, it is easy to do as a first pass and sometimes provides satisfactory results (see lower left hand corner of Figure 5).

The program `moon.py` filters the images shown in Figure 5. As in the previous example, we ask you to supply the missing lines which perform the brick wall filtering (if you can't figure this out you can check the lines in the program `convolve_fft.py` on the website.

5 Spectrograms

A spectrogram is a plot of the power spectrum as a function of time. For each time (indicated on the x-axis) the variance of the power spectrum as a function of frequency is plotted along the y-axis. A color or gray scale is used to indicate the magnitude of the variance. In this way a 3-D plot is plotted in 2-D.

Spectrograms are used to plot the power spectra describing the evolution of a time-varying process. Perhaps the most common uses of a spectrogram are to analyze spoken words and the calls of animals, such as bird songs. Other applications includes the analysis of music, sonar, radar and seismology.

Here we illustrate the use of a spectrogram by looking at a bird song. Most often bird songs are stored in a *.wav file. A *.wav file is the standard Microsoft audio file format. There are several Internet sites from which *.wav files can be downloaded for analysis, for example,

`http://www.ilovewavs.com/Effects/Birds/Birds.htm`

Bird songs are also recorded in `mp3` format. In this case the mp3 files can be converted to *.wav files.

There are two practical problems. First, in our experience, *.wav file downloaded from the Internet are often corrupted. Fortunately such files can often be restored by using a software package called Audacity which can be freely downloaded. Moreover, a useful U-tube video explains how to use Audacity to recover a corrupted *.wav file

`http://www.youtube.com/watch?v=zaVr_WLikqs`

The second problem is that Python currently has no audio playback software. Therefore in order to hear the birdsong you will need to use a Quick time player

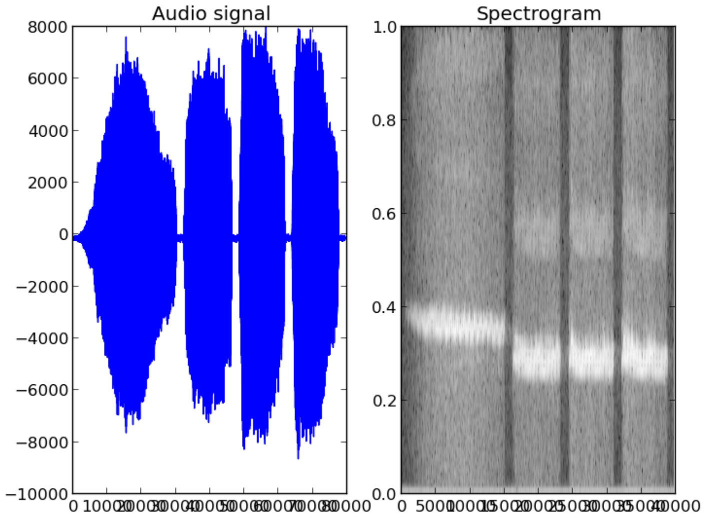


Figure 6: Spectrogram for a goldfinch.

or some other software package to hear the bird song (Windows Media Player (PCs) or Quick Time (PCs and Mac's).

Figure 6 shows the birdsong of a goldfinch downloaded from the above bird song website. The bird calls out twice. By looking at the spectrogram alone, can you guess what the call sounds like? The Python program that generates this spectrogram is called `spectro_bird.py`. Modify this program so that the generated spectrogram uses a color scale instead of a gray scale.

Deliverables: Use `Lab9_template.tex` to prepare the lab assignment.

References

[1] J. S. Bendat and A. G. Piersol. *Random Data: Analysis and Measurement Procedures, Second Edition*. John Wiley & Sons, New York, 1986.

[2] D. J. Field. Rotation between the statistics of natural images and the response properties of cortical cells. *J. Opt. Soc. Amer. A*, 4:2379–2394, 1987.

[3] D. J. Field and N. Brady. Visual sensitivity, blur and the sources of variability in the amplitude spectra of natural scenes. *Vision Res.*, 37:3367–3382, 1997.

[4] V. Fugère and R. Krahe. Electric signals and species recognition in the wave–type gymnotiform fish *apteronotus leptorhynchus*. *J. Exp. Biol.*, 213:225–236, 2010.

[5] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes: The art of scientific computing, third edition*. Cambridge University Press, New York, 2007.

[6] D. L. Ruderman. Origins of scaling in natural images. *Vision Res.*, 37:3385–3398, 1977.

[7] D. L. Ruderman and W. Bialek. Statistics of natural images: scaling in the woods. *Phys. Rev. Lett.*, 73:814–817, 1994.

[8] D. J. Tolhurst, Y. Tudmor, and T. Chao. Amplitude spectra of natural images. *Ophthalm. Physiol. Optics*, 12:229–232, 1992.