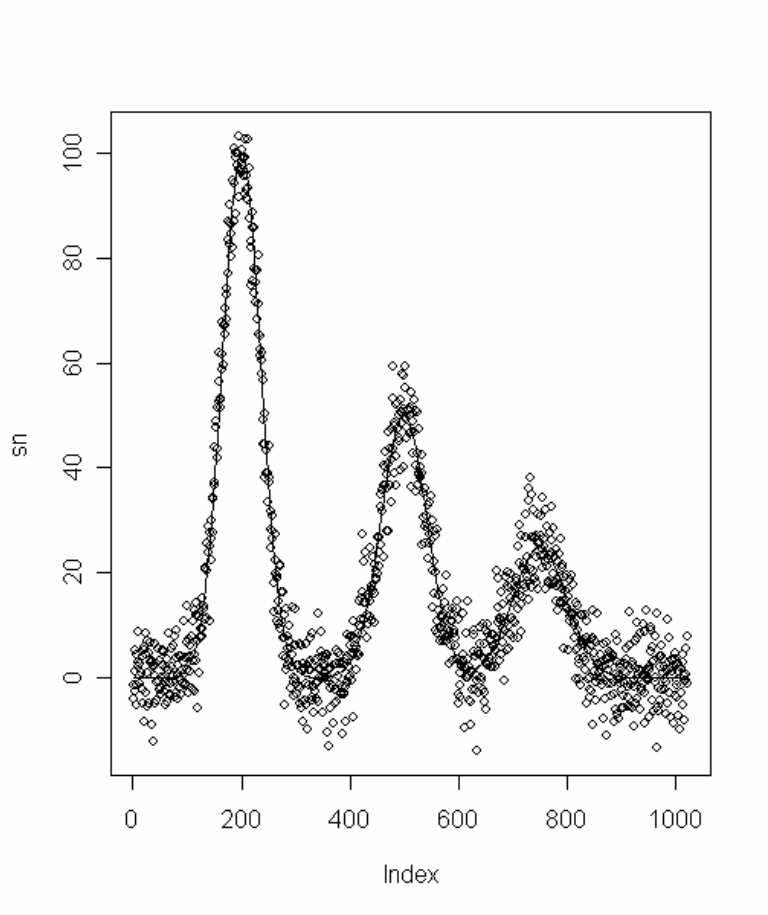
Using R for Smoothing and Filtering

In the following handout words and symbols in **bold** are R functions and words and symbols in *italics* are entries supplied by the user; underlined words and symbols are optional entries (all current as of version R-2.4.1). Sample texts from an R session are highlighted with gray shading.

The data in this tutorial are in the file "Signal&Noise," which consists of a true signal of three Gaussian peaks, pure homoscedastic noise and the raw signal, which is a sum of the true signal and the noise. Each object has 1024 discrete values. The figure below shows the raw signal as discrete points and the true signal as a smooth curve.

```
> plot(sn)                                    # raw signal plus as points
> points(signal, type = "l")                  # line represents true signal
```



Moving-Average and Savitisky-Golay Smoothing Functions

R provides a generic function for smoothing data that uses a user-defined moving-average or Savitsky-Golay smoothing function. The basic syntax for the command is:

filter(*data***,** *filtercoef***)**

where *data* is an object containing the data being smoothed and *filtercoef* is an object containing the smoothing function's coefficients. To create a 5-point moving-average smoothing function, for example, we can use the following command:

```
> ma5 = c(1, 1, 1, 1, 1)/5; ma5      # vector of 1s divided by size of function gives…

 [1] 0.2  0.2  0.2  0.2  0.2              #...a vector of coefficients whose sum is 1
```

For a small moving-average smoothing function, this is a reasonable approach; however, for a large-moving average smoothing function, such as one involving 25 points, this is tedious. We can use the function **rep** to simplify the code

rep(*value***,** *repetitions***)**

where *value* is the value to be repeated and *repetitions* is the number of repetitions. A 25-point moving-average smoothing function can be created with the following code

```
> ma25 = c(rep(1, 25))/25; ma25

 [1] 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04
[12] 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04
[23] 0.04 0.04 0.04
```

A Savitsky-Golay smoothing function is just as easy to code; thus, for a 5-point quadratic or cubic smoothing function we have

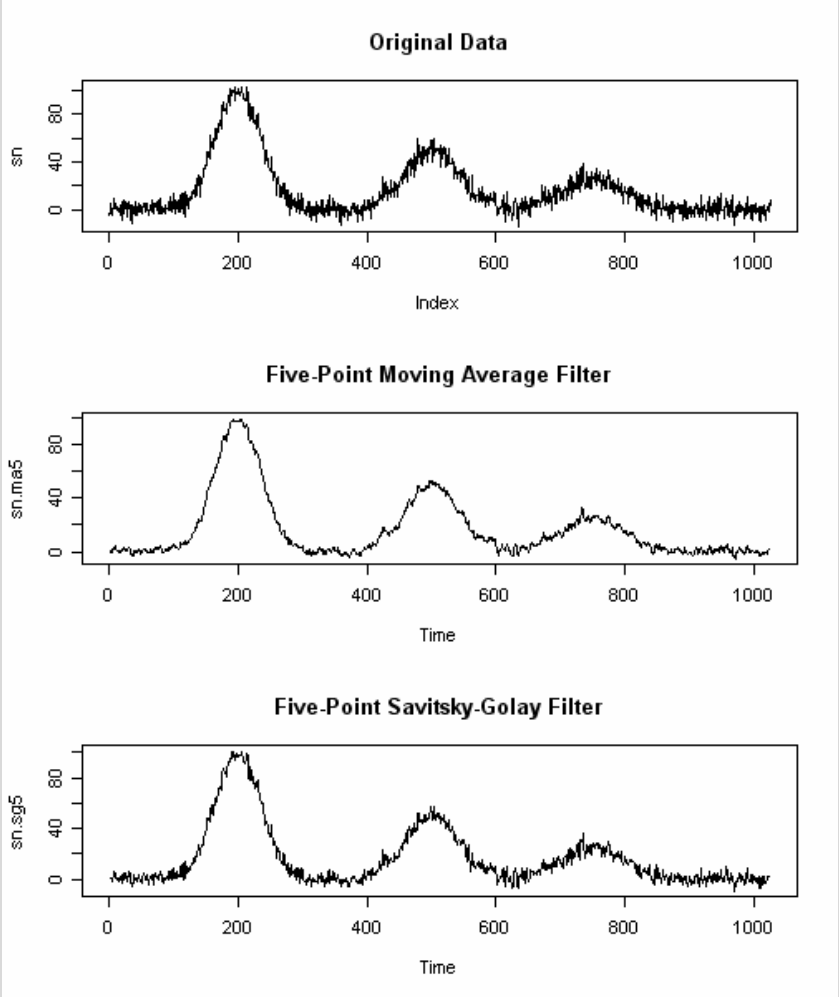
```
> sg5 = c(-3, 12, 17, 12, -3)/35
```

where the terms within the code c( ) are the coefficients, and the divisor is the normalization constant. Values for these coefficients and normalization constant come from published tables.

As a practical example, the following code

```
> layout(matrix(1:3, 3,1))                    # set up three areas for graphing
> plot(sn,type="l",main="Original Data")
> sn.ma5=filter(sn,ma5)                       # apply five-point moving average filter
> sn.sg5=filter(sn,sg5)                       # apply five-point Savitsky-Golay filter
> plot(sn.ma5, main="Five-Point Moving Average Filter")
> plot(sn.sg5, main="Five-Point Savitsky-Golay Filter")
```

produces a much improved signal



Fourier Filtering

Two commands are used when Fourier transforming data. To obtain the Fourier transform the command is

fft(*object***)**

where *object* is a vector containing the data. To obtain the inverse Fourier transform, the code is

fft(*object***, inverse = TRUE)**

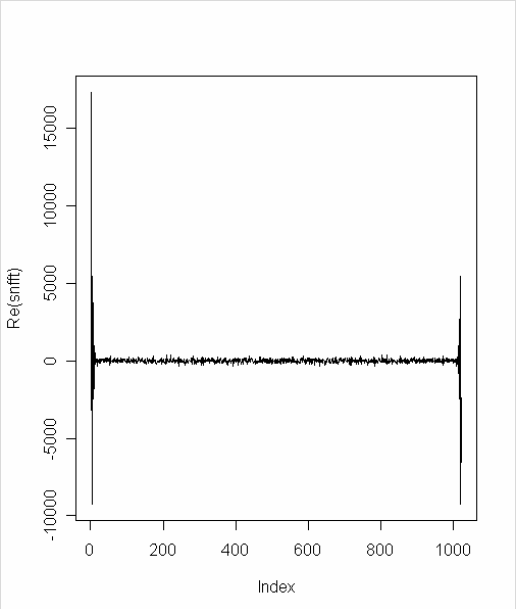
Taking the Fourier transform of a vector followed by the inverse Fourier transform should return the original vector. Because the command **fft** does not return a normalized vector, however, it is necessary to divide the inverse Fourier transform by the vector's length; thus

```
> t=1:4; t                                          # object with values of 1, 2, 3 and 4

[1] 1  2  3  4

> t1 = fft(t); t1                                   # taking the FFT of t produces…

 [1] 10+0i  -2+2i -2+0i -2-2i                       # …this set of four complex numbers

> t2 = fft(t1, inverse = TRUE); t2                  # taking the inverse FFT produces…

 [1]  4+0i  8+0i 12+0i 16+0i                        # …a result that is not equivalent to t
but…

> t3 = fft(t1, inverse = TRUE)/length(t1); t3       # dividing the IFFT by length returns…

[1] 1+0i 2+0i 3+0i 4+0i                             # …the original t
```

The Fourier transform, as shown above, creates a vector of complex numbers consisting of real and imaginary parts. The real part of the complex number provides information about the amplitudes and frequencies in terms of cosine functions and the imaginary part provides the same information in terms of sine functions. We can work with either, but the convention is to use the real terms. To extract the real part of the Fourier transform we use the command

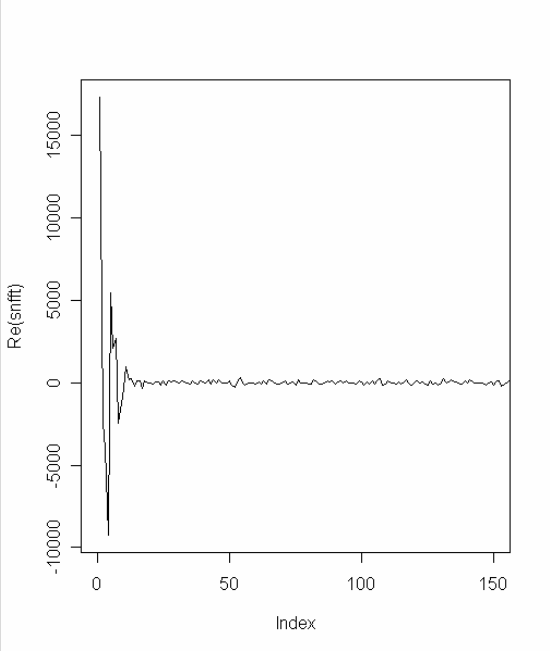
Re(*object***)**

```
> sn.fft = fft(sn); plot(Re(sn.fft), type = "l")        # plot the real part of FT only
```



This plot is symmetrical (nearly) because only the first 512 points are unique; the remaining 512 points are a mirror image, within computational limitations. The Fourier transform cannot produce additional information; since we start with 1024 points and end with both real and imaginary points, each can have only 512 unique values. To separate the real terms into those due to the signal and those representing noise, we expand the first portion of the data; thus

```
> plot(Re(sn.fft), xlim = c(0, 150), type = "l")          # examine the first 150 points
```



By the 50th point we seem comfortably into terms accounting for noise only. To filter the noise we set all but the first 50 and the last 50 points to zero; thus

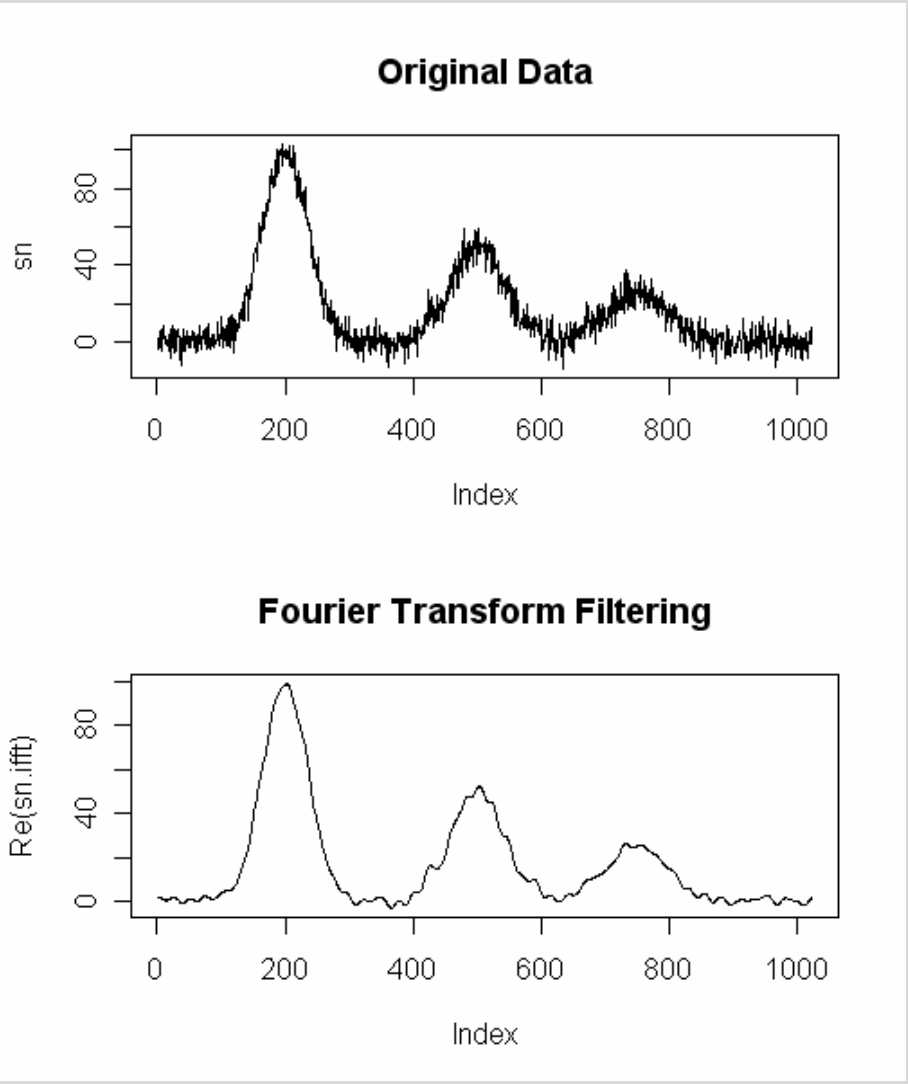
```
> sn.fft[51:974] = 0 + 0i
```

Note that both the real and imaginary parts are set to zero. Next, we complete the normalized inverse Fourier transform

```
> sn.ifft = fft(sn.fft, inverse = TRUE)/length(sn.fft)
```

Finally, we plot the filtered and original data, using the real portion of the sn.ifft (note – the function fft always produces a complex number, but for the inverse fft the imaginary portion is zero and only the real portion contains information)

```
> layout(matrix(1:2,2,1)
> plot(sn, type="l", main="Original Data")
> plot(Re(sn.ifft),type="l", main="Fourier Transform Filtering")
```


Original Data
sn
0
40
80
0
200
400
600
800
1000
Index
Fourier Transform Filtering
Re(sn.ifft)
0
40
80
0
200
400
600
800
1000
Index