EE355/GP265 Handout # ~~9~~ 10

<more from Chapter 3 in the text>

Practical Issues in Data Processing

We have seen that we can describe a radar echo from a patch on the ground as ~~a~~ convolution of the actual surface reflectivity and the transmitted waveform:

$$s(t) = r(t) * b(t)$$

where $s(t)$ is the received signal, $r(t)$ is the transmitted (reference) signal, and $b(t)$ is the true image brightness distribution. We also found that correlation of the signal with the reference obtains an image $i(t)$ that approximates $b(t)$, and is in fact the convolution itself of $b(t)$ with the autocorrelation of $r(t)$:

$$i(t) = s(t) \star r(t)$$

$$= b(t) * \left(r(t) \star r(t) \right)$$

We will denote the impulse response of the radar $h_{radar}(t) = r(t) \star r(t$

But how do we actually implement these?

Correlator algorithms (range)

Very often the range processor (or entire processor) of a radar system is called the correlator system, because its job is to correlate the received echoes with known reference functions. Time-domain convolution or correlation according to standard definitions would work, but is quite slow when implemented on a digital computer. So instead we usually use frequency-domain implementations of the convolution theorem.

Refresher - the convolution theorem

Recall the convolution theorem:

$$a * b \underset{\text{Fourier transform}}{\Longleftrightarrow} A \cdot B$$

or, simply, convolution in the time domain is equivalent to multiplication in the frequency domain. There is also a converse: $a \cdot b \Leftrightarrow A * B$.

Correlation is much the same:

$$a \circledast b \underset{\text{F.T.}}{\Longleftrightarrow} A^{*} B$$

In the practical world we use fft routines to implement the Fourier transforms.

FFT routines

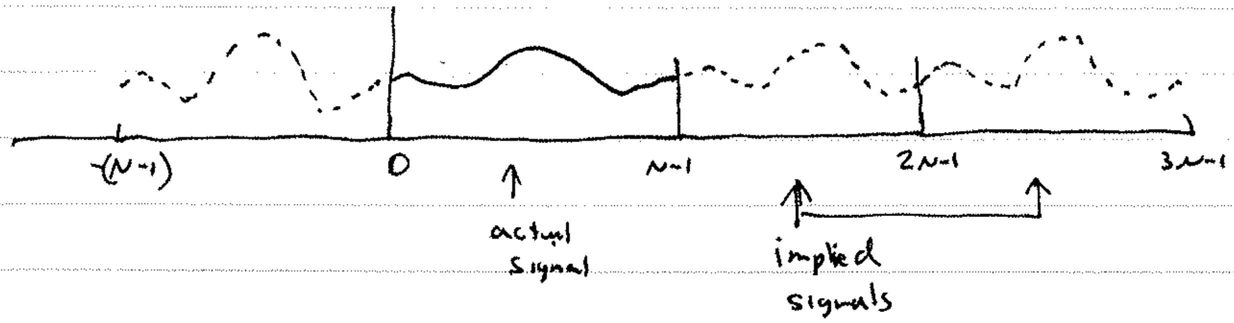
For the purposes of our class we simply consider the FFT as a fast implementation of a discrete Fourier transform. The main differences between discrete and continuous transforms relevant to our problem are:

- the data are finite length and discretely sampled
- all transforms are circular in geometry, an artifact of the discrete ~~n~~ nature of the data

	Point Spacing	Total Extent
Time domain:	$1/f_s$	N/f_s
Frequency domain:	f_s/N	f_s

where f_s is the sample rate and N is the FFT length.

Circular interpretation



One consequence of this is that the FFT enforces continuity across the boundaries, possibly inducing a large jump in the signal.

FFT parameters and calling:

We typically call an FFT routine in a program like

call fft (data, size, direction) (Fortran example)

where data is a complex data array, size is the length in complex samples (power of 2), and direction is ±1 for forward/reverse transforms. Note conventions differ on whether 1 is forward or reverse. You may download the fft algorithm from the fortran numerical recipes package from anonymous ftp login on jakey.stanford.edu. Look in the EE355 directory.

<u>Example of use - spectrum of a discrete chirp signal</u>

Let's write a simple program to plot the spectrum of a discretely sampled chirp signal. Our chirp parameters are

slope $(s) = 10^{12}$ Hz s^{-1}

pulse length $(\tau) = 10\ \mu s$ (10^{-5})

$f_s = 100$ MHz (10^{8})

Analytically we can define this signal as

$$r(t) = \exp\left(j\left[\pi \cdot s \cdot t^2\right]\right) \qquad -\frac{\tau}{2} < t < \frac{\tau}{2}$$

In the discrete world, we need to ~~cons~~ consider the length of the pulse in <u>points</u>, not just seconds. The number of points n_{pts} is the product of time and sample rate:

$$n_{pts} = \tau \cdot f_s$$
$$= 10^{-5} \times 10^{8} = 10^{3} \quad \text{for our chirp above}$$

Hence we can describe the chirp by the complex sequence

$$r_i = \exp\left(j \cdot \pi \cdot s \cdot \left[i \cdot \frac{1}{f_s}\right]^2\right), \quad -\frac{n_{pts}}{2} < i < \frac{n_{pts}}{2}$$

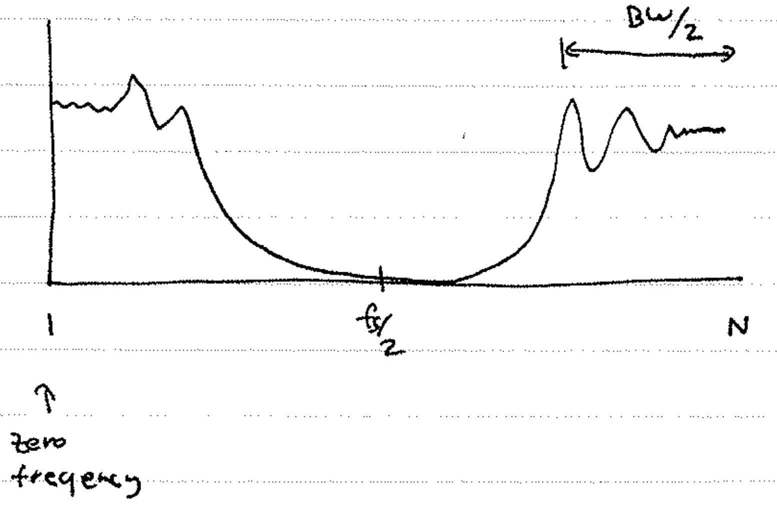
Note that here we have also the inconvenience of ~~of~~ negative indices, which are a problem in a computer sometimes. Hence we often offset data in arrays to solve this, but then we have to remember to shift the result back to compensate for this.

So lets write a pseudo-program to calculate the spectrum of a digital chirp, using Fortran-like constructs.

```
       pi = 3.14159265
       s = 1.0 e+12
       tau = 10.0 e-6
       fs = 100.0e6
c- create chirp signal
       npts = fs * tau
       do i = -npts/2, npts/2
          t = i/fs
          phase = pi*s* t**2
          ref(i+npts/2+1) = cexp (cmplx (0, phase))
      end do
c- calculate transform
      call fft (ref, 2048, 1)
c- write out magnitude in dB
      do i = 1, 2048
         write (file,*) 20 * alog10 (cabs(ref(i))+1.e-30)
      end do
      end
```

Note we wrote out the data in dB format, and added 10^{-30} to each point in case we had a zero value in ref.

The output from this would look like



What would we expect the bandwidth to be?

$$BW = s \cdot \tau$$
$$= 10^{12} \cdot 10^{-5} = 10^{7}\ Hz$$

whereas the full spectrum has bandwidth

$$BW_{full} = f_s = 10^{8}\ Hz$$

Hence $\approx 10\%$ of the spectrum is filled with signal.

<u>A practical range-compression algorithm</u>

Now let's look at implementing a range compression algorithm in a way that is reasonably efficient. First, what are our logical steps:

1) Create a reference signal once and store it for repeated use.
2) Create a loop processing each line one at a time.
3) Read in the data, copy it into complex form.
4) Transform the data to the frequency domain
5) Cross multiply with conjugate of reference
6) Inverse transform the product
7) Write out the results.

A little more detail:

Since we will be using the same reference signal over and over, we only need create it once to save computations. In fact, we will repeatedly use the transform of the reference, so we can also create a transformed version to save one transform in our loop.

Our pseudocode might look like:

```
c- create a reference signal
      do i = -npts/2, npts/2
         t = i/fs
         phase = pi*s*t**2
         ref( i+npts/2+1 ) = cexp(cmplx (0, phase))
      enddo
c- transform reference
      call fft( ref, N, 1)
c- begin loop to process each line
      do line = 1, nlines
         read (file) signal
         call fft( signal, N, 1)
         do i = 1, length
            signal(i) = signal(i) * conjg(ref(i))
         end do
         call fft (signal, N, -1)
         write (outfile) signal
      end do
```

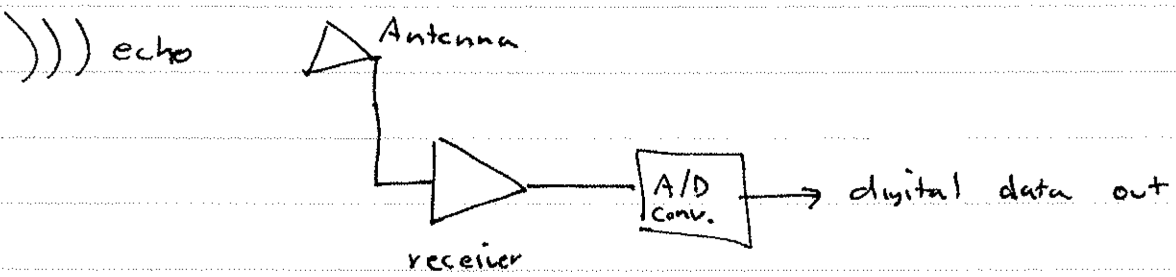
Raw data formats

All of the above assumes that the signal data are initially stored in the same complex floating point format that the computer uses. For several reasons this is not usually the case: doing so would be exceptionally wasteful of space, and the data would have to be reformatted for each type of computer representation.

Usually the samples of the received echo are digitized and stored as byte (8-bit) values.

However, the data quantizer in the radar may not use all 8 bits. In the ERS-1 case, for example, the data are quantized to 5 bits. In JERS-1, the data are quantized to 3 bits.

What do we mean by all this?

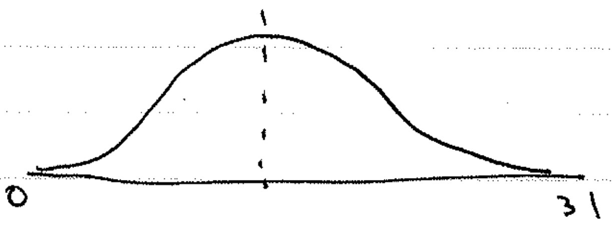


For a 5-bit quantizer, the available output levels are 0, 1, 2, .. 31. For a 3-bit quantizer, they range from 0 to 7. These values are usually stored as one byte each for distribution, although occasionally only the bits actually used are stored.

Since a floating point number uses about 4 bytes of storage, this results in a savings of a factor of 4.

The quantizer is adjusted so that the ~~average~~ output of the A/D converter is near the middle of the range for zero volts input. That way as the input voltage swings positive and negative both can be accommodated.

If we were to look at a histogram of ERS data, say, we would see the following:



Data number in byte

The mean of this distribution for ERS-1 is about 15.5. Hence we can relate the output voltage from the receiver V_r to the data number (dn) by

$$V_r = A \cdot (dn - 15.5)$$

where A is a scaling constant that gives the gain of the receiver / quantizer.

In addition, ERS data are quantized directly in complex format (called I/Q) so that each pair of samples represents one complex data value. In other words, we can translate from raw data to complex format using

$$\text{signal}_i = \text{cmplx}\,(\text{raw}_{i*2-i} - 15.5,\ \text{raw}_{i*2} - 15.5)$$

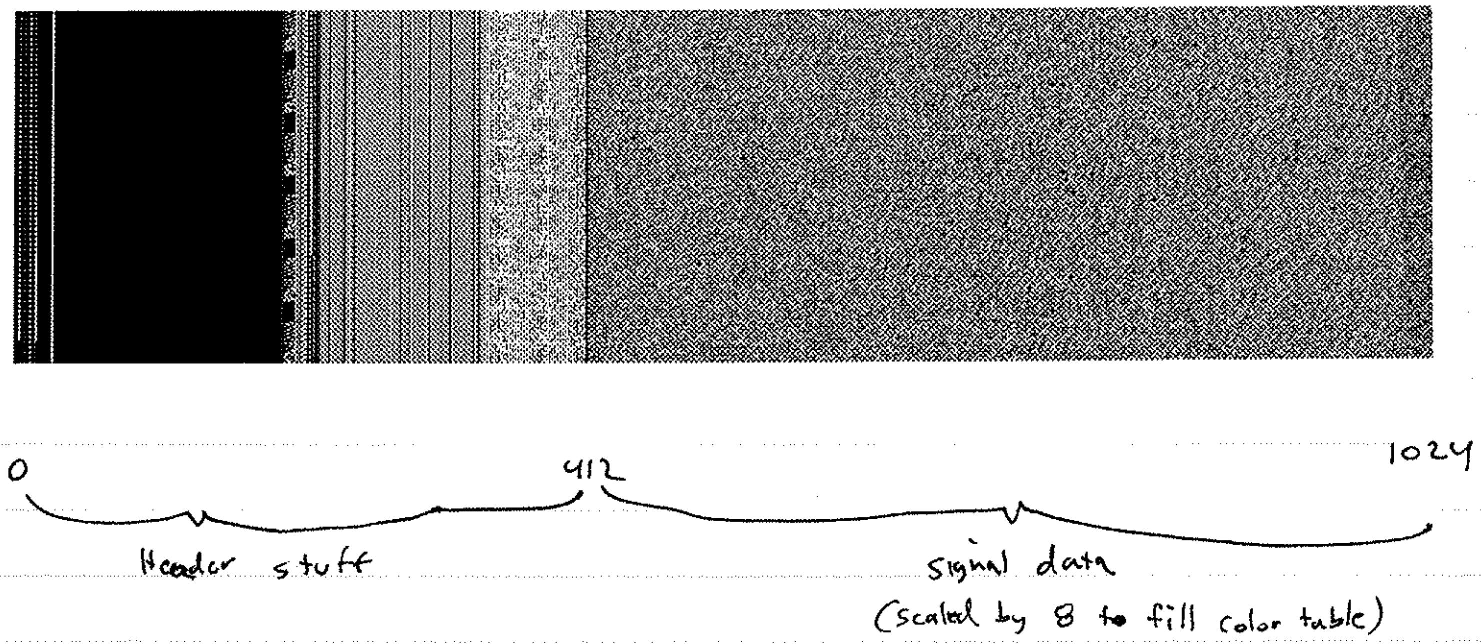
where $signal_i$ is the ith sample, raw_i is the ith raw byte in the line, and cmplx(,) converts two reals to a complex value.

<u>Header bytes</u>

One other aspect of raw data worth mentioning is that quite often ancillary data are stored along with the signal data in each line to aid further processing. This can be as simple as a line counter or it can include detailed encoded information about the radar parameters and imaging geometry.

In the ERS case, each data line is preceded by 412 bytes of descriptive data. Much of this simply repeats from line to line, while the rest of it changes as each line increments. The existence of repeating headers also allows you to solve for the line length if you don't know it.

A typical ERS display looks like:


0
412
1024
Header stuff
signal data
(scaled by 8 to fill color table)