Course Notes for Engineering 100 Music Signal Processing Revised March 2013 College of Engineering The University of Michgan

Professor Andrew E. Yagle
Dept. of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122

Chapter 1

Introduction to Musical Signals and to Engineering 100 Technical Material

1.1 Introduction to the Course

These are a set of course notes for the freshman engineering course Engineering 100 (Music Signal Processing) at the University of Michigan, Ann Arbor. No prior knowledge of calculus is assumed; the highest level of mathematics used consists of logarithms and the cosine addition formula. However, some very basic familiarity with Matlab is assumed (the required level of familiarity is provided in Lab #1). No programming skill is needed or provided; the entire course can be completed without using a programming loop or a conditional statement.

The goal of this course is to provide students with the engineering background necessary to analyze and synthesize simple musical signals. In the first half of the course, three labs teach students the necessary skills. These labs are:

1. Basic Matlab skills (mostly plotting)
2. Determining musical frequencies:
    - Using a simple digital signal processing (DSP) algorithm to compute frequencies present in a tonal version of "The Victors" (the Michigan fight song);
    - Interpreting the computed frequencies using log-log and semi-log plots;
    - Inferring the existence of accidental notes (sharps and flats) from these log-log and semi-log plots.
3. Fourier series, spectrogram and spectra:
    - Computing the spectra of various signals using the FFT;
    - Filtering out noise using the FFT;
    - Visualizing "The Victors" plus some interference using a spectrogram;
    - Using this spectrogram and the FFT to remove the interference.

Also during the first half of the course, students complete two one-week projects:

1. Tone synthesizer and transcriber:
    - Programming in Matlab a simple tone synthesizer that generates tonal music from an on-screen keyboard;

- Programming in Matlab a simple tone transcriber that accepts a tonal music signal from the synthesizer and outputs a pseudo-musical notation using Matlab's `stem` command.

2. Reverse-engineer touch-tone phone signals:
   - Computing the spectra of touch-tone phone signals using the FFT;
   - Programming in Matlab a touch-tone synthesizer that generates touch tones from an on-screen keyboard;
   - Programming in Matlab a touch-tone transcriber that accepts a touch-tone signal and outputs the phone number;
   - Analyzes the performance (error rate vs. SNR) of the transcriber in noise.

During the second half of the course, students program in Matlab a simple music synthesizer that outputs any of four instruments (guitar, trumpet, clarinet, and one generated by the students using additive synthesis) playing whole, half, or quarter notes in a one-octave range, from an on-screen keyboard. Reverb for the trumpet signal is also included. Students also program in Matlab a music transcriber that accepts musical signals from the synthesizer and outputs a pseudo-musical notation using Matlab's `stem` command. Performance of the transcriber, with noise, as measured by error rate, is evaluated.

It should be noted that technical communication and working in teams constitute half of the course grade, and are prominent in grading the final project. Ethical issues involved in copyrighting and sampling digital music are also addressed. Important as they are, these notes focus on only the technical material of the course. **Technical communication is covered in a separate textbook.**

1.2 Musical Signals

A signal is a function of time that carries information. A musical signal is a function of time that represents the output of a musical instrument playing musical notes. Polyphonic music and vocals are beyond the scope of this course. What one hears with their ears can be considered an auditory signal, consisting of time-varying changes of air pressure. However, we shall assume that the music has been played into a microphone and converted into an electrical signal.

- I will use `this font` for Matlab commands;
- I will give the Matlab code used for all plots;
- You can copy Matlab programs directly from the .pdf file of these notes to the Matlab command window, to save retyping.

1.2.1 Tonal Signals

The simplest type of musical signal is a **pure tone**, or **tonal signal**. This is simply a sinusoidal signal; when listened to, it is perceived as a simple tone. A tonal signal of note B looks like

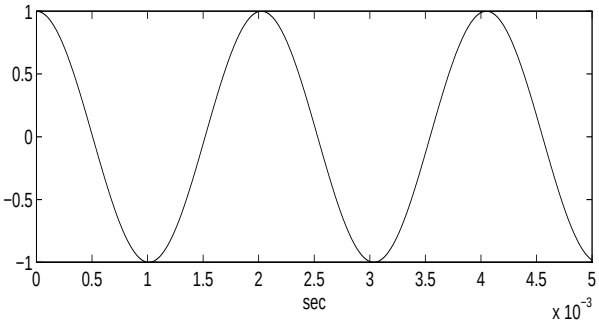


Figure 1.1: Tonal note B signal

It is easy to see that this is a sinusoid with a period of about 0.0020 seconds (2.0 ms), so that

its frequency is about $\frac{1}{0.0020}$=500 Hertz. We will discover in Lab #2 that the actual frequency of a tonal note B is 494 Hertz.

This plot was generated in Matlab using

```
T=linspace(0,0.005,1000);
X=cos(2*pi*494*T);
subplot(211),plot(T,X)
```

To listen to a tonal B signal, run Matlab and type in the following at the Matlab prompt >> (you will need earphones in a CAEN lab):

```
T=linspace(0,1,10000);
X=cos(2*pi*494*T);soundsc(X,10000)
```

Tonal music isn't very interesting, although it is easy to analyze. We will use this fact in Chapter 2 to compute the frequencies of the musical tones in a tonal version of "The Victors." When analyzing a phenomenon, such as music, it is usually a good idea to start with the simplest possible form of the phenomenon, gain some understanding of it, and then proceed to more complicated forms of the phenomenon.

1.2.2 Trumpet Signal

The musical signal of a trumpet playing note B:

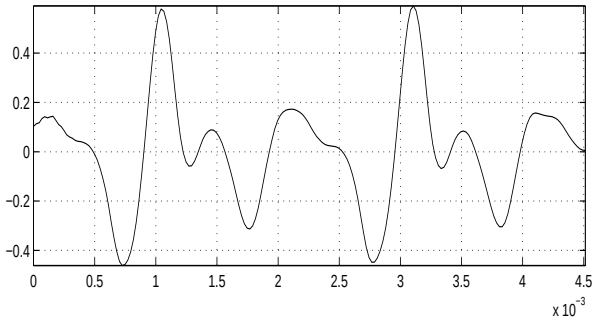


Figure 1.2: Trumpet playing note B

This signal is clearly much more complicated than the pure tonal signal, but one feature stands out: it is clearly periodic with a period of about 0.0020 seconds (2.0 msec), just like the pure tone signal. The period of the signal specifies the note played, whatever the instrument.

This plot was generated in Matlab using

```
load trumpet.mat;
T=linspace(0,32767/44100,32768);
subplot(211),plot(T(1:200),X(1:200))
```

To listen to the trumpet, run Matlab, download the file `trumpet.mat` from the course web site, copy it to the working directory and type

```
load trumpet.mat;soundsc(X,44100)
```

Although the trumpet signal looks complicated, we will see in Chapter 3 that it is actually well-approximated by the sum of nine sinusoids at frequencies that are multiples of 494 Hertz. This makes it easy to analyze and to synthesize.

1.3 Sinusoidal Signals

It is already clear that sinusoids will play a central role in the study of musical signals. The basic form of a sinusoidal signal is

$$x(t) = A\cos(2\pi f t + \theta) \tag{1.1}$$

where we define the following constants:

- A=amplitude=the maximum value of $x(t)$;
- f=frequency in Hertz of $x(t)$;
- θ=phase (shift) in radians of $x(t)$;
- $P = \frac{1}{f}$=period in seconds of $x(t)$;
- $x(t) = x(t + P)$ for all times t.

The **period** of any signal is the amount of time P before it repeats. If the signal is not periodic, then the period $P \to \infty$. **The musical signal generated by a musical instrument playing a single note indefinitely is periodic.**

The sinusoid $x(t)$ is periodic with the period $P = \frac{1}{f}$ since (using $fP = 1$)

$$\begin{aligned} x(t+T) &= A\cos(2\pi f(t+P)+\theta) \\ &= A\cos(2\pi ft + \theta + 2\pi fP) \\ &= A\cos(2\pi ft + \theta + 2\pi) \\ &= A\cos(2\pi ft + \theta) = x(t). \quad (1.2) \end{aligned}$$

This is why we need the 2π multiplying f.

The phase shift θ can be interpreted as a time delay (shift to the right) of $-\frac{\theta}{2\pi f}$ since

$$A\cos(2\pi ft + \theta) = A\cos\left(2\pi f\left[t + \frac{\theta}{2\pi f}\right]\right).$$

However, we will not spend any more time on phase shifts since a sinusoid delayed by a small amount of time sounds just like the same sinusoid without the delay. Phase shift is important when adding sinusoids at the *same* frequency, but not when adding sinusoids at *different* frequencies. Phase shift affects $x(t)$ greatly, but not how the human ear perceives it, as you will discover in Chapter 3 (some music scientists disagree; they think humans can "hear phase").

1.4 Sampling Signals

Musical signals are inherently functions $x(t)$ of continuous time t, where t is a member of the set of real numbers. This makes them unsuitable for processing on digital computers (analog computers are hard to find outside museums). So it is necessary to convert musical signals into sequences of numbers that can be stored and processed on computers. This is done by **sampling**.

To **sample** a signal $x(t)$, we simply take its values at times t that are integer multiples $\frac{n}{S}$, of some small number $\frac{1}{S}$, where S is the **sampling rate** in $\frac{\text{SAMPLE}}{\text{SECOND}}$ and n is an integer. The result is a sequence of numbers, which we label $x[n]$. This sequence of numbers $x[n]$ is what we store and process on a digital computer.

You can think of the physical act of sampling a continuous-time voltage signal $x(t)$ as closing a switch for an instant every $\frac{1}{S}$ seconds, or at a rate of S times per second, and storing those values of $x(t)$. Or you can think of the physical act of sampling as multiplying $x(t)$ by a train of very narrow pulses separated by $\frac{1}{S}$ seconds. The pulse at time $t = \frac{n}{S}$ is multiplied by $x(\frac{n}{S})$. All other values of $x(t)$ are multiplied by zero. The result of this multiplication is a train of pulses of separated by $\frac{1}{S}$ seconds, where the height of the n^{th} pulse is $x(\frac{n}{S}) = x[n]$.

Let the sinusoid $x(t) = \cos(2\pi 1000t)$ be sampled at $8000\frac{\text{SAMPLE}}{\text{SECOND}}$. The result is

$$\begin{aligned} x[n] &= x\left(t = \frac{n}{8000}\right) \\ &= \cos\left(2\pi 1000\frac{n}{8000}\right) \\ &= \cos\left(2\pi n\frac{1000}{8000}\right) \\ &= \cos([\pi/4]n). \quad (1.3) \end{aligned}$$

The sequence of numbers $x[n]$ is

n	...	0	1	2	3	4	...
x[n]	...	1	.71	0	–.71	–1	...

The act of sampling this sinusoidal signal is illustrated in Figure 1.3. The heights of the circles indicate the values of $x[n]$, and the stems connect them to the times $t = \frac{n}{S}$ at which they are sampled. Only heights and times are known.

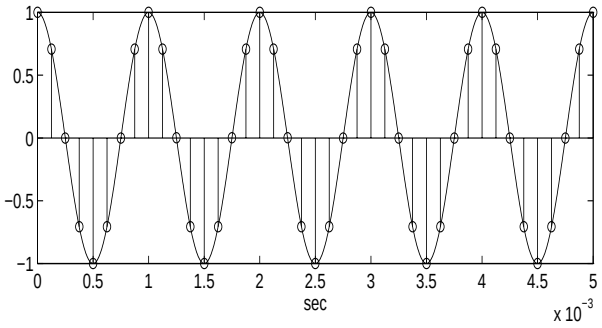


Figure 1.3: Sampling a sinusoidal signal

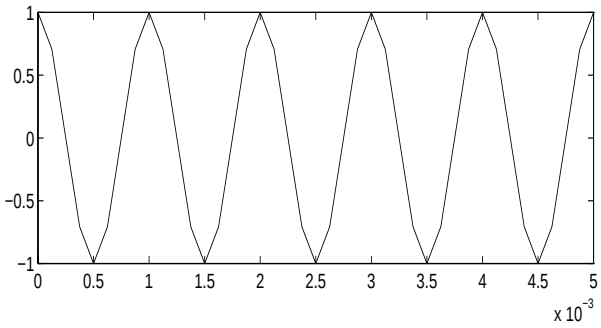


Figure 1.4: Linear interpolation of a 1000 Hertz sinusoid sampled at $8000\frac{\text{SAMPLE}}{\text{SECOND}}$.

This plot was generated in Matlab using

```
T1=linspace(0,0.005,1000);
X1=cos(2*pi*1000*T1);
subplot(211),plot(T1,X1)
hold on %Overlay two plots
T2=linspace(0,0.005,41);
X2=cos(2*pi*1000*T2);
subplot(211),stem(T2,X2)
```

The number of points to use was computed as $(8000\frac{\text{SAMPLE}}{\text{SECOND}})(0.005 \text{ sec.})$=40 samples. Since both 0 and 0.005 seconds were plotted, we used 40+1=41 samples (try counting them yourself).

1.5 Reconstruction of Signals from Their Samples

A major issue is how to reconstruct the continuous-time signal $x(t)$ from its samples $x[n]$. How can we go from the sequence of numbers $x[n] = \{\ldots 1, .71, 0, -.71, -1 \ldots\}$ back to $x(t) = \cos(2\pi 1000t)$? We can try "connecting the dots" with straight lines; see Figure 1.4.

This operation is called *linear interpolation*. That does not look very much like a sinusoid!

This plot was generated in Matlab using

```
T=linspace(0,0.005,41);
X=cos(2*pi*1000*T);
subplot(211),plot(T,X)
```

Matlab's `plot` "connects the dots" with straight lines, i.e., performs linear interpolation.

If we sample the sinusoid faster, at $40000\frac{\text{SAMPLE}}{\text{SECOND}}$, then linear interpolation gives

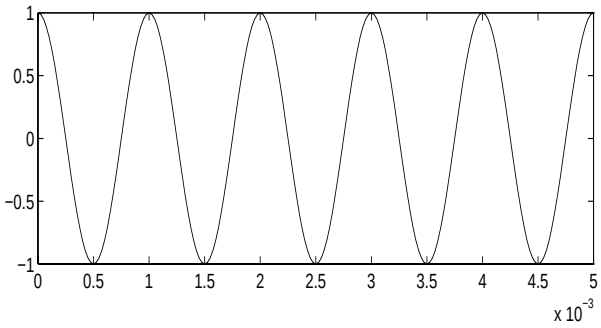


Figure 1.5: Linear interpolation of a 1000 Hertz sinusoid sampled at $40000\frac{\text{SAMPLE}}{\text{SECOND}}$.

Figure 1.5 does look more like a sinusoid.

The number of points to use was computed as $(40000\frac{\text{SAMPLE}}{\text{SECOND}})(0.005 \text{ sec.})$=200 samples. Since both 0 and 0.005 seconds were plotted, we used 200+1=201 samples, making a smooth curve.

In fact, in Chapter 3 we will derive a remarkable theorem called the *sampling theorem*. This theorem states that if the sampling rate $S\frac{\text{SAMPLE}}{\text{SECOND}}$ exceeds **double** the maximum frequency of the signal $x(t)$ (which we will define in Chapter 3), then $x(t)$ can be reconstructed **exactly** from its samples $x[n]$! Here, as long as we know that the maximum frequency of the sinusoid is 1000 Hertz, then sampling faster than just $2000\frac{\text{SAMPLE}}{\text{SECOND}}$ allows $x(t)$ to be reconstructed from its samples $x[n]$. This will not be done by "connecting the dots" with straight lines!

1.6 Two Useful Trig Identities

Finally, we derive two trig identities that we will use over and over in this course. These are the **only** trig identities we will need!

Recall cosine addition and subtraction laws

$$\begin{aligned}\cos(X+Y) &= \cos(X)\cos(Y) - \sin(X)\sin(Y)\\ \cos(X-Y) &= \cos(X)\cos(Y) + \sin(X)\sin(Y).\end{aligned}$$

Adding these gives

$$\cos(X+Y) + \cos(X-Y) = 2\cos(X)\cos(Y). \tag{1.4}$$

This identity allows us to convert the product of two cosines into the sum of two cosines. We will use this formula extensively.

The other trig identity is obtained by setting $X = 2\pi f$ and $Y = \theta$ in the cosine *subtraction* formula and then multiplying by A. This gives

$$\begin{aligned}A\cos(2\pi ft-\theta) &= A\cos(2\pi ft)\cos(\theta) \qquad (1.5)\\ &+ A\sin(2\pi ft)\sin(\theta)\\ &= C\cos(2\pi ft) + B\sin(2\pi ft)\end{aligned}$$

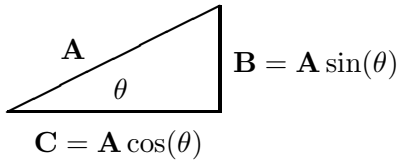
where B and C are defined as

$$\begin{aligned}B &= A\sin(\theta)\\ C &= A\cos(\theta). \qquad (1.6)\end{aligned}$$

From the following picture, it is clear that

$$\begin{aligned}A &= \sqrt{B^2+C^2}\\ \theta &= \arctan[B/C]. \qquad (1.7)\end{aligned}$$

"Arctan" is the arctangent or inverse tangent.



As an example, this identity states that

$$5\cos(2\pi ft)+12\sin(2\pi ft) = 13\cos(2\pi ft-\arctan[2.4]).$$

since $\sqrt{5^2+12^2} = 13$ and $\frac{12}{5} = 2.4$. Note the sign of the phase is negative. Also note that if $C < 0$ add 180^o to the phase, since the output of the arctan function is restricted to $|\theta| < 90^o$.

To see why we need this, this formula gives $-\cos(2\pi ft) = \cos(2\pi ft + 0)$ unless we add 180^o to the phase since $C = -1 < 0$.

This formula shows that a cosine with a phase shift is equal to the sum of a pure sine and a pure cosine, all at the same frequencies. It will prove useful in Chapter 3.

Chapter 2

Computing and Interpreting the Frequencies of a Sampled Sinusoid

2.1 Overview

In this chapter we lay the groundwork necessary to compute and interpret the frequencies present in the sampled tonal version of "The Victors." This includes both computation of frequencies, and interpretation of the frequencies computed.

1. We will derive a simple digital signal processing (DSP) algorithm for computing the frequency of a sinusoid from three consecutive samples of the sampled sinusoid;

2. We will examine the issues in the implementation of this simple algorithm (formula);

3. We will derive the concepts of log-log and semi-log plots for interpretation of data. These will be useful in many fields of engineering, not just music signal processing;

4. We will see how missing data values can be discerned and computed. This will allow us to infer the existence of accidental (sharp and flat) notes, even though they are not present in "The Victors."

2.2 Computing Frequency of a Sinusoid from its Samples

2.2.1 Derivation of the Formula

We are given samples $x[n] = x(t = \frac{n}{S})$ of a continuous-time sinusoid sampled at $S\frac{\text{SAMPLE}}{\text{SECOND}}$. We do not know the frequency, amplitude, or phase of the sinusoid. All we know is:

- $x(t)$ **is** a sinusoid; and
- Its frequency $f < \frac{S}{2}$=half sampling rate.

The latter condition is the sampling theorem requirement that we sample faster than twice the maximum (here, the only) frequency of $x(t)$. Here we will get a first glimpse of where that condition comes from, and why we need it.

Since $x(t)$ is a sinusoid, we know that its samples $x[n]$ are the sequence of numbers

$$\begin{aligned} x[n] &= x\left(t = \frac{n}{S}\right) \\ &= A\cos\left(2\pi f\frac{n}{S} + \theta\right) \\ &= A\cos\left(2\pi\frac{f}{S}n + \theta\right). \end{aligned} \tag{2.1}$$

Replacing n with $n+1$ throughout gives

$$\begin{aligned} x[n+1] &= A\cos\left(2\pi\frac{f}{S}(n+1)+\theta\right) \\ &= A\cos\left[\underbrace{2\pi\frac{f}{S}n+\theta}_{X}+\underbrace{2\pi\frac{f}{S}}_{Y}\right] \\ &= A\cos(X+Y). \end{aligned} \tag{2.2}$$

Replacing n with $n-1$ throughout gives

$$\begin{aligned} x[n-1] &= A\cos\left(2\pi\frac{f}{S}(n-1)+\theta\right) \\ &= A\cos\left[\underbrace{2\pi\frac{f}{S}n+\theta}_{X}-\underbrace{2\pi\frac{f}{S}}_{Y}\right] \\ &= A\cos(X-Y). \end{aligned} \tag{2.3}$$

Now substitute the expressions defined above:

- $X = 2\pi\frac{f}{S}n+\theta$ and $Y = 2\pi\frac{f}{S}$

into the trig identity (1.4) from Chapter 1

$$2\cos(X)\cos(Y) = \cos(X+Y)+\cos(X-Y).$$

and then multiply by A. This gives

$$\begin{aligned} & 2A\cos\left(2\pi\tfrac{f}{S}n+\theta\right)\cos\left(2\pi\tfrac{f}{S}\right) \\ = \quad & A\cos\left(2\pi\tfrac{f}{S}(n+1)+\theta\right) \\ + \quad & A\cos\left(2\pi\tfrac{f}{S}(n-1)+\theta\right). \end{aligned} \tag{2.4}$$

Looking at each term, we see this becomes

$$2x[n]\cos\left(2\pi\frac{f}{S}\right) = x[n+1]+x[n-1] \tag{2.5}$$

which can be rearranged into the formula

$$\boxed{f = \tfrac{S}{2\pi}\arccos\left[\tfrac{x[n+1]+x[n-1]}{2x[n]}\right]} \tag{2.6}$$

where "arccos" is the arccosine or inverse cosine.

This is clearly a formula for computing the frequency of a sinusoid from any three consecutive samples $\{x[n-1], x[n], x[n+1]\}$ and the sampling rate $S\frac{\text{SAMPLE}}{\text{SECOND}}$ used to obtain those samples.

2.2.2 Example Use of the Formula

Let us use this formula on the sampled sinusoid in Chapter 1. Recall that sampling a sinusoid at $S = 8000\frac{\text{SAMPLE}}{\text{SECOND}}$ resulted in

n	...	0	1	2	3	4	...
x[n]	...	1	.71	0	–.71	–1	...

Setting $n = 1$ in the formula gives

$$\begin{aligned} f &= \frac{S}{2\pi}\arccos\left[\frac{x[1+1]+x[1-1]}{2x[1]}\right] \\ &= \frac{8000}{2\pi}\arccos\left[\frac{0+1}{2(0.71)}\right] \\ &= \frac{8000}{2\pi}\arccos[0.71] \\ &= \frac{8000}{2\pi}\frac{\pi}{4} = 1000 \quad \text{Hertz.} \end{aligned} \tag{2.7}$$

which was the frequency of the sinusoid.

You will use this formula in Lab #2 to compute the frequencies of the tones in a tonal version of "The Victors."

2.2.3 Problems with the Formula

One obvious problem arises if we set $n = 2$ in the formula. This gives

$$\begin{aligned} f &= \frac{S}{2\pi}\arccos\left[\frac{x[2+1]+x[2-1]}{2x[2]}\right] \\ &= \frac{8000}{2\pi}\arccos\left[\frac{.71-.71}{2(0)}\right]. \end{aligned} \tag{2.8}$$

We obtain the indeterminate form $\frac{0}{0}$! In real life we would seldom be so unlucky as to sample

the sinusoid at a time at which it is identically zero. However, if $\{x[n-1], x[n], x[n+1]\}$ are all very small, numerical issues (roundoff) can cause problems in computing the arccosine.

Of course, in real life we would have far more than three samples available. Then we use the formula on each triplet $\{x[n-1], x[n], x[n+1]\}$ of samples. Most of the computed frequencies will be identical; the few that are different due to roundoff error can be discarded. Of course, we should ensure that roundoff error is the cause of the different values, called *outliers*.

Another issue arises if the sampling rate S is too small, specifically, if $S < 2f$. To see why this is a problem, suppose we sample the 7000 Hertz sinusoid $y(t)$=cos($2\pi 7000t$) at the rate S=8000$\frac{\text{SAMPLE}}{\text{SECOND}}$. The samples are sequence

$$\begin{aligned} y[n] &= y\left(t = \frac{n}{8000}\right) \\ &= \cos\left(2\pi 7000 \frac{n}{8000}\right) \\ &= \cos\left(2\pi n \frac{7000}{8000}\right) \\ &= \cos([7\pi/4]n). \end{aligned} \tag{2.9}$$

The sequence of numbers $y[n]$ is

n	...	0	1	2	3	4	...
y[n]	...	1	.71	0	–.71	–1	...

But these are the same numbers as in the previous example for f=1000 Hertz! Indeed,

$$\begin{aligned} x(t) = \cos(2\pi 1000t) &\neq y(t) = \cos(2\pi 7000t) \\ x[n] = \cos([\pi/4]n) &= y[n] = \cos([7\pi/4]n) \end{aligned}$$

since sampled signal $y[n]$ can be rewritten as

$$\begin{aligned} y[n] &= \cos([7\pi/4]n) \\ &= \cos([8\pi/4]n - [\pi/4]n) \\ &= \cos(-[\pi/4]n) = \cos([\pi/4]n) \\ &= x[n] \end{aligned} \tag{2.10}$$

since cos(X) is periodic with period 2π and even.

So a 7000 Hertz cosine and a 1000 Hertz cosine are identical after sampling at 8000$\frac{\text{SAMPLE}}{\text{SECOND}}$. The formula assumes that the actual cosine is at 1000 Hertz, the frequency which is less than half the sampling rate. So it will give the wrong answer if this isn't true, i.e., $f > S/2$.

A more subtle issue occurs if the sampling rate is too large! Then $x[n]$ is slowly varying, so that $x[n-1] \approx x[n] \approx x[n+1]$ and $\frac{x[n+1]+x[n-1]}{2x[n]} \approx 1$. The formula requires computation of the arccosine of a number near one, and again roundoff error can cause problems. To see why, examine a zoom of cos(X) for tiny X:

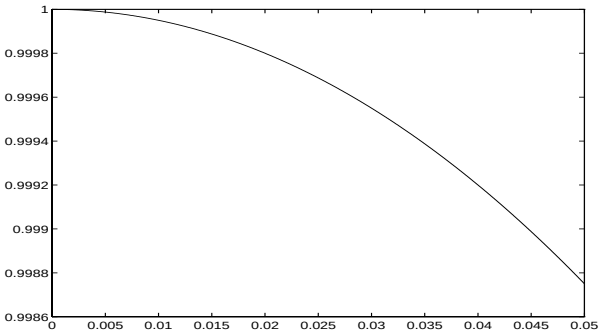


Figure 2.1: cos(X) for tiny values of X

Computing the arccosine of 0.999 requires solving cos(X)=0.999. It can be seen that a small change in 0.999 will cause a big change in X=arccos(0.999), and therefore in the computed frequency f. A smaller sampling rate avoids this.

Finally, the formula was derived under the assumption that $x(t)$ is a pure sinusoid. If there is any noise added to $x(t)$, this is no longer true, and the formula breaks down (in fact, it breaks down quite badly!). In Chapter 3 we will learn a much better way than using this formula to compute frequencies from samples of signals.

2.3 Tuning a Piano

As an aside, we note that trig identity (1.4) shows how to tune a piano.

Suppose we desire to tune a specific piano key to note A (440 Hertz). The piano key is actually tuned to 442 Hertz. How can we correct the piano key to 440 Hertz if we don't know any of:

- What note A sounds like (we are tone-deaf);
- We want to tune the piano to 440 Hertz;
- The piano is actually tuned to 442 Hertz;
- What "Hertz" even means or what trig is.

We can derive a solution to this as follows.

Set $X{=}2\pi 441t$ and $Y{=}2\pi t$ in (1.4). This gives

$$\cos(2\pi 440t){+}\cos(2\pi 442t){=}2\cos(2\pi t)\cos(2\pi 441t).$$

The left side is the sum of the desired signal (the correctly tuned piano) and the actual signal (the actual mistuned piano). The right side is a sinusoid at the frequency halfway between the desired and actual frequencies, multiplied by the *slowly-time-varying amplitude* $2\cos(2\pi t)$:

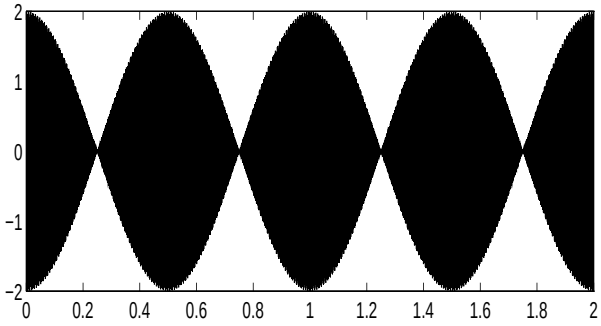


Figure 2.2: $\cos(2\pi 440t) + \cos(2\pi 442t)$.

The variation of 441 Hertz shows up as solid black since the plot lines are all squeezed together. But the amplitude is varying slowly.

This suggests a procedure for tuning a piano, if we have a tuning fork or sinewave generator that generates a tone at the desired frequency:

- Strike the tuning fork and mistuned piano key simultaneously;
- The result will sound like a tone with a slowly-varying amplitude;
- Tighten or loosen the piano string so that the variation gets slower;
- When the variation has stopped, the piano is correctly tuned.
- Try yourself: `T=linspace(0,5,100000); X=cos(2*pi*440*T)+cos(2*pi*442*T); soundsc(X,20000)`
- You can do this even if you are tone-deaf!

"You can Tune a Piano, But You Can't Tuna Fish" (an early REO Speedwagon album).

2.4 Interpreting Data

2.4.1 Problem Statement

Suppose we have run an experiment and we obtained the following data:

n	1	2	3	4	5
x[n]	2	4	8	16	32

Our goal is to interpret this data-what rule or formula is generating it? (Likely you already know the answer, but this is a deliberately simple example, so that you can follow it easily.)

The first thing to do is plot the data, since the human mind interprets pictures well, finding patterns in them. Drawing a picture is a good first step in many mathematical, scientific or engineering problems. The result is Figure 2.3.

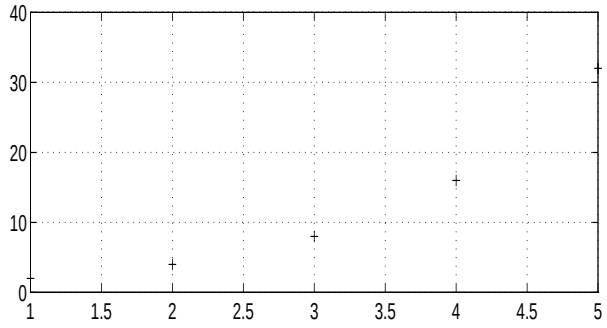


Figure 2.3: Plot of x[n] vs. n.

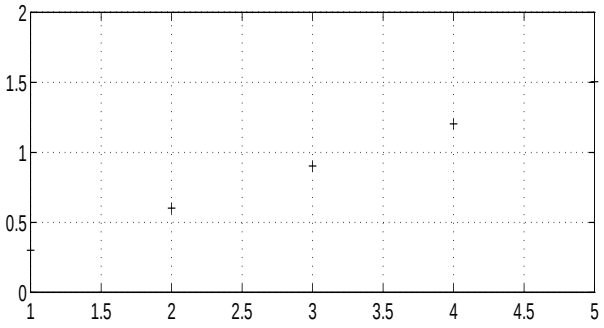


Figure 2.4: Semi-log plot of x[n] vs n.

This plot was generated in Matlab using

```
X=[2 4 8 16 32];
subplot(211),plot(X,'+'),grid on
```

This looks like a parabolic or exponential curve, but which is it? Is it a quadratic, cubic, or something in between? Is the base of the exponential two, three, or something else? How can we answer all of these questions quickly?

2.4.2 Semi-Log Plots

Suppose the formula is exponential, so that $x[n] = ba^n$ for some constants a and b. Taking the logarithm (base 10) of this equation gives

$$\begin{aligned} x[n] &= ba^n \qquad (2.11) \\ \log_{10}(x[n]) &= n\log_{10}(a) + \log_{10}(b). \end{aligned}$$

This means that if we plot $\log_{10}(x[n])$ vs. n, we will get a straight line with slope $\log_{10}(a)$ and y-intercept (where the line crosses the vertical $n = 0$ axis) $\log_{10}(b)$. So plotting $\log_{10}(x[n])$ vs. n is a quick way to see whether the formula is exponential, and if it is to find the values of a and b. This is called a **semi-log** plot since we only take the logarithm of $x[n]$, not of n.

Figure 2.4 is a plot of $\log_{10}(x[n])$ vs. n:

This plot was generated in Matlab using

```
X=[2 4 8 16 32];
plot(log10(X),'+'),grid on
```

- This is clearly a straight line, so the formula is exponential;
- The slope is $\frac{1.5-0.3}{5-1}$=0.3.
- Hence $\log_{10}(a) = 0.3 \rightarrow a = 10^{0.3} = 2$.
- Note the y-intercept is **not** 0.3, since 0.3=$\log_{10}(x[1])$, not $\log_{10}(x[0])$. Extend the line to n=0. Then the y-intercept is zero.
- Since we know the formula is $x[n] = b2^n$, it is easier to skip the y-intercept and simply plug in $b = \frac{x[1]}{2^1} = 1$.
- The formula is therefore $x[n] = 1(2)^n$.

2.4.3 Log-Log Plots

Suppose we run a different experiment and we obtain the following data:

n	1	2	3	4	5
y[n]	1	4	9	16	25

A plot of $y[n]$ vs n looks very much like Fig. 2.3, so it is not shown. But a semi-log plot of $y[n]$ vs. n isn't a straight line! What do we do?

Suppose the formula is polynomial, so that $y[n] = bn^a$ for some constants a and b. Taking the logarithm (base 10) of this equation gives

$$\begin{aligned} y[n] &= bn^a \qquad (2.12) \\ \log_{10}(y[n]) &= a\log_{10}(n) + \log_{10}(b). \end{aligned}$$

This means that if we plot $\log_{10}(y[n])$ vs. $\log_{10}(n)$ (**not** n), we will get a straight line with slope a (**not** $\log_{10}(a)$) and y-intercept $\log_{10}(b)$. So plotting $\log_{10}(y[n])$ vs. $\log_{10}(n)$ is a quick way to see whether the formula is polynomial, and if it is to find the values of a and b. This is called a **log-log** plot, since we take the logarithms of **both** $y[n]$ and n (two logarithms, not one).

Figure 2.5 is a plot of $\log_{10}(y[n])$ vs. $\log_{10}(n)$:

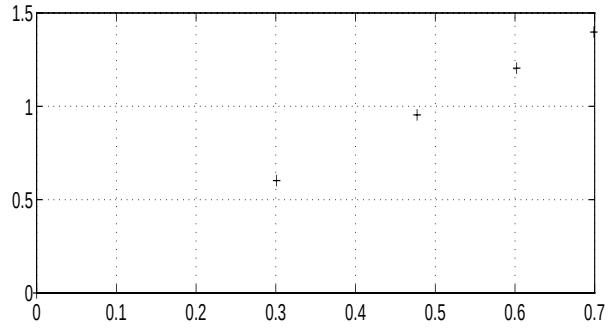


Figure 2.5: Log-log plot of y[n] vs n.

This plot was generated in Matlab using

```
N=[1:5];Y=[1 4 9 16 25];
plot(log10(N),log10(Y),'+'),grid on
```

- This is clearly a straight line, so the formula is polynomial, not exponential, in form;
- The slope is $\frac{1.4-0.0}{0.7-0.0}$=2.0. Hence $a = 2.0$;
- The y-intercept is 0.0 (n=0 is in the plot);
- Hence $0.0 = \log_{10}(b)$, and $b = 10^{0.0} = 1$;
- The formula is therefore $y[n] = 1(n)^{2.0}$.

2.4.4 Semi-Log and Log-Log Scales

We can save the trouble of computing and interpreting logarithms by plotting the data directly using semi-log and log-log axis scales. Before calculators were invented, it was a big time-saver to be able to plot the data directly, without looking up logarithms in a table (as I remember well).

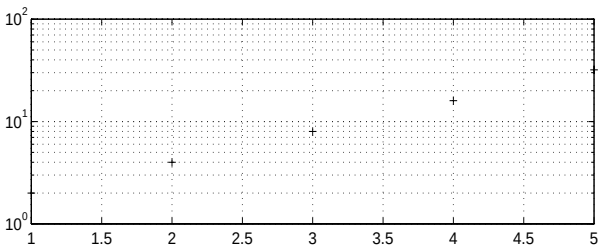


Figure 2.6: Plot of x[n] vs n on Semi-Log Scale.

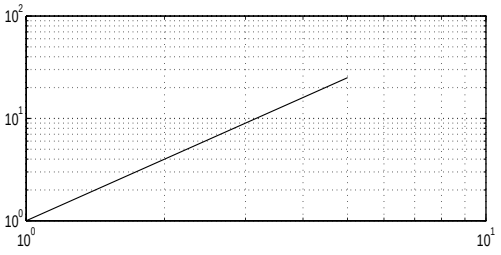


Figure 2.7: Plot of y[n] vs n on Log-Log Scale.

These plots were generated in Matlab using

```
X=[2 4 8 16 32];
N=[1:5];Y=[1 4 9 16 25];
subplot(211),semilogy(X,'+'),grid on
subplot(211),loglog(N,Y),grid on
```

A straight line is used for the log-log plot since it is too hard to see the crosses otherwise.

2.4.5 Missing Single Value

Now we run still another experiment and obtain the following data:

x[n]	2	4	8	32	64	128

Note we do not specify the values of n. The reason is that we don't know them! All we know is that the six values of n are a subset of the seven values $\{1, 2, 3, 4, 5, 6, 7\}$, but we don't know which values! In other words, there is a missing $x[n]$, but we don't know the missing value or where it belongs! What can we do?

A semi-log plot of $x[n]$ is illuminating:

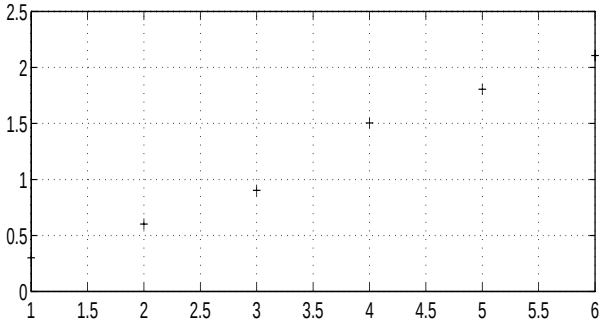


Figure 2.8: Semi-Log Plot of x[n].

This plot was generated in Matlab using

```
X=[2 4 8 32 64 128];
plot(log10(X),'+'),grid on
```

Note the following about this plot:

- The first three points lie on a straight line with slope=$\frac{0.9-0.3}{3-1} = 0.3$;
- The third and fourth points lie on a straight line with slope=$\frac{1.5-0.9}{4-3} = 0.6$;
- The last three points lie on a straight line with slope=$\frac{2.1-1.5}{6-4} = 0.3$;
- The second slope is exactly **twice** the first and third slopes.

Of course, any two points lie on a straight line, but the slope values cannot be coincidence.

If there were a missing value between the third and fourth points, they would be twice as far apart, and the slope of the line connecting them would match the other two slopes. That is,

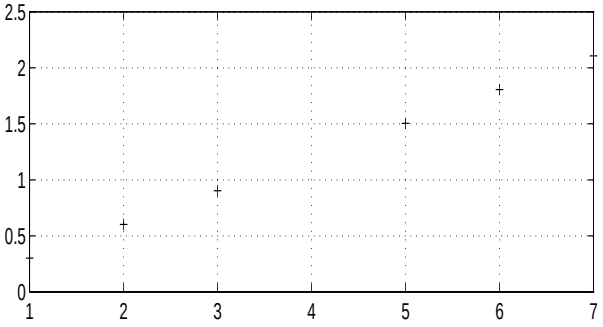


Figure 2.9: Semi-Log Plot with Missing Value.

This plot was generated in Matlab using

```
X=[2 4 8 32 64 128];
N=[1 2 3 5 6 7];
plot(N,log10(X),'+'),grid on
```

Figure 2.9 makes it clear that:

- The missing value is indeed at $n = 4$;
- The missing value is about $\log_{10}(x[4]) = 1.2$;
- The missing value is then $x[4] = 10^{1.2} = 16$;

- The formula is therefore $x[n] = 1(2)^n$.

We have discerned the existence, location, and value of a missing data point!

2.4.6 Missing Multiple Values

Now we run one last experiment and obtain the following data:

y[n]	1	4	16	32	128	512

A semi-log plot of $y[n]$ is illuminating:

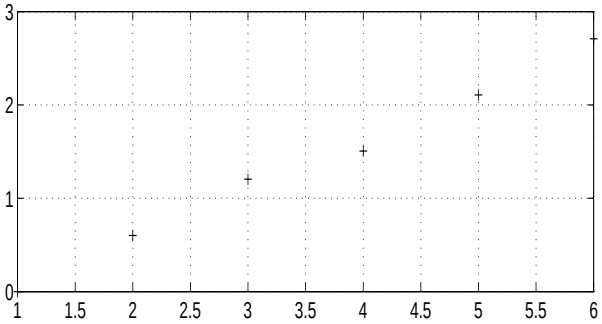


Figure 2.10: Semi-Log Plot of y[n].

This plot was generated in Matlab using

```
Y=[1 4 16 32 128 512];
plot(log10(Y),'+'),grid on
```

Note the following about this plot:

- The first three points lie on a straight line with slope=$\frac{1.2-0}{3-1} = 0.6$;
- The third and fourth points lie on a straight line with slope=$\frac{1.5-1.2}{4-3} = 0.3$;
- The last three points lie on a straight line with slope=$\frac{2.7-1.5}{6-4} = 0.6$;
- The second slope is exactly **half** the first and third slopes.

This is like the previous example, except that the middle slope is half, not double, the first and third slopes. This means that the only place where there **isn't** a missing value is between 16 and 32! Inserting gaps between **all other** pairs of points **except** 16 and 32 gives Figure 2.11:

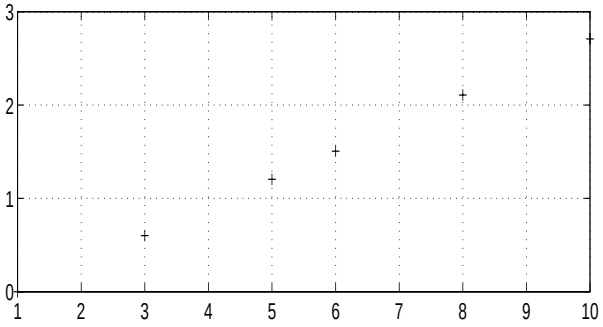


Figure 2.11: Semi-Log Plot with Missing Values.

This plot was generated in Matlab using

```
Y=[1 4 16 32 128 512];
N=[1 3 5 6 8 10];
plot(N,log10(Y),'+'),grid on
```

The formula is $y[n] = 2(2)^n$. We have discerned the existence, locations, and values of many missing data points!

In Lab #2 you will use this concept to infer the existence of accidental notes (sharps and flats), although they are not present in "The Victors."

Chapter 3

Fourier Series and Musical Signals

3.1 Overview

In this chapter we introduce the concept of the Fourier series expansion of a periodic signal. Any real-world periodic signal can be written as a sum of sinusoids whose frequencies (in Hertz) are integer multiples of the reciprocal of the period (in seconds). The amplitudes and phases of the sinusoids are all that is needed to specify the Fourier series expansion.

Musical signals are a natural application of Fourier series, since musical signals (specifically, an instrument playing a specific note) are periodic. The sinusoid with the same period as the music signal is called the *fundamental*; it is the pure tone that sounds most like the signal.

To obtain the richer sound of an instrument playing a note, we add in *harmonics* (musicians call them *overtones*). Harmonics are sinusoids at frequencies double, triple, etc. the frequency of the fundamental. The amplitudes of the harmonics determine the *timbre* (sound) of the instrument playing the note. The reason a violin playing note B sounds different from a trumpet playing note B is that the amplitudes of the harmonics are different in the trumpet from the violin.

Why should you care about Fourier series?

1. The Fourier series representation of a musical signal is specified by only a few numbers (the amplitudes and phases of the harmonics, and the note);

2. Fourier series show that a continuous-time signal can be reconstructed from its samples, provided the sampling rate exceeds double the maximum frequency of the harmonics (the sampling theorem);

3. Fourier series show how a noisy signal can be *filtered* to remove almost all of the noise, even if the noise drowns out the signal.

3.2 Fourier Series

We will make no attempt to prove the existence of Fourier series. This topic is properly handled in a graduate-level math course. There is quite a history, as well as a lot of math, behind this topic. But if we are willing to accept the *existence* of a Fourier series expansion of musical signals, we can derive everything else we need.

3.2.1 Basics of Fourier Series

Let $x(t)$ be a periodic real-world continuous-time signal with period P, so that $x(t) = x(t+P)$ for

all times t. Then $x(t)$ can be written as a (usually infinite) sum of sinusoids with frequencies that are integer multiples of $\frac{1}{P}$ Hertz. This is called a **Fourier series**. We can write

$$x(t) = \sum_{k=0}^{\infty} c_k \cos\left(2\pi \frac{k}{P} t - \theta\right). \qquad (3.1)$$

There are periodic functions $x(t)$ that cannot be so expanded, but they have bizarre properties and are of interest only to mathematicians.

In practice, the infinite series is truncated to a finite number K of terms, giving a **finite Fourier series**. Finite Fourier series still give good approximations to most periodic signals. The larger K is, the better the approximation.

A simple example of a finite Fourier series is

$$\begin{aligned} x(t) &= \frac{\cos(2\pi[1]t)}{1} + \frac{\cos(2\pi[3]t)}{9} + \frac{\cos(2\pi[5]t)}{25} \\ &+ \frac{\cos(2\pi[7]t)}{49} + \frac{\cos(2\pi[9]t)}{81}. \end{aligned} \qquad (3.2)$$

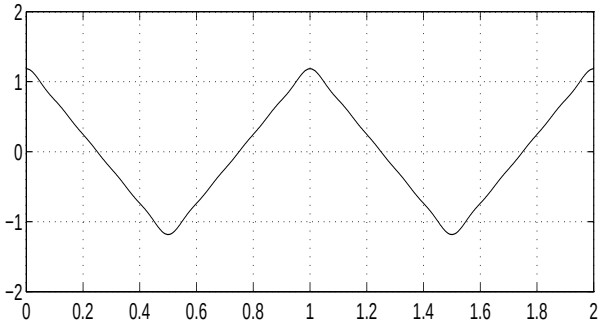


Figure 3.1: Finite Fourier Series Approximation of a Triangle Wave with Period=1 second.

This plot was generated in Matlab using

```
T=linspace(0,2,1000);
X=zeros(1,1000);for K=1:2:9;
X=X+cos(2*pi*K*T)/K/K;end
subplot(211),plot(T,X)
```

$x(t)$ is clearly a good approximation to a triangle wave with period=1, except for the slight rounding at the corners of the triangles. Note that including more terms in the Fourier series would sharpen these rounded corners.

To compute Fourier series coefficients such as $\{\frac{1}{1}, \frac{1}{9}, \frac{1}{25}, \frac{1}{49}, \frac{1}{81} \ldots\}$ analytically, you must take a course in Fourier analysis such as EECS 216. To compute them numerically, read on...

3.2.2 Fourier Series of a Trumpet

Since instruments playing musical notes create periodic signals, musical signals have Fourier series expansions. The Fourier series can be truncated to a finite number of terms and still do a good job of representing the musical signal. For example, recall the trumpet signal in Chapter 1:

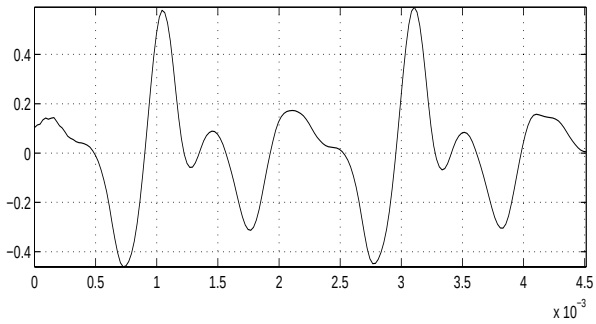


Figure 3.2: Actual Trumpet Signal.

The trumpet is playing note B. The signal is clearly periodic with a period of about 0.0020 seconds. In fact, we know from Lab #2 that note B has a frequency of 494 Hertz, so the period of note B is $P = \frac{1}{494}$ seconds. Therefore, the trumpet signal can be expanded as a Fourier

series which is a sum of sinusoids at frequencies of $\{494, 2(494), 3(494)\ldots\}$ Hertz. Using values

$c_0 = 0.0000$	$\theta_0 = +0.0000$ radians
$c_1 = 0.1155$	$\theta_1 = -2.1299$ radians
$c_2 = 0.3417$	$\theta_2 = +1.6727$ radians
$c_3 = 0.1789$	$\theta_3 = -2.5454$ radians
$c_4 = 0.1232$	$\theta_4 = +0.6607$ radians
$c_5 = 0.0678$	$\theta_5 = -2.0390$ radians
$c_6 = 0.0473$	$\theta_6 = +2.1597$ radians
$c_7 = 0.0260$	$\theta_7 = -1.0467$ radians
$c_8 = 0.0045$	$\theta_8 = +1.8581$ radians
$c_9 = 0.0020$	$\theta_9 = -2.3925$ radians

the nine-term finite Fourier series approximation to the trumpet signal is plotted in Figure 3.3:

$$\begin{aligned} x(t) &= \underbrace{c_0}_{\text{DC}} + \underbrace{c_1\cos(2\pi(1)494t-\theta_1)}_{\text{FUNDAMENTAL}} \\ &+ \underbrace{c_2\cos(2\pi(2)494t-\theta_2)}_{\text{HARMONIC}} + \ldots \\ &= \sum_{k=0}^{9} c_k\cos(2\pi(k)494t-\theta_k). \quad (3.3) \end{aligned}$$

We will compute c_k and θ_k in Section 3.4.2.

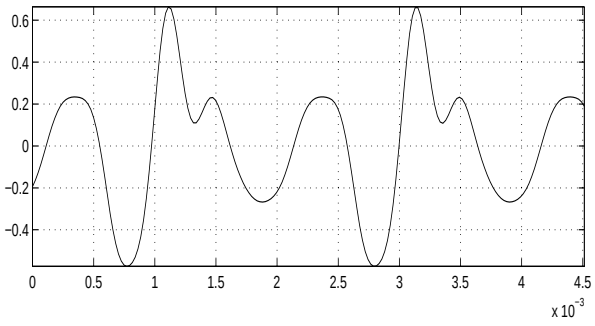


Figure 3.3: Synthetic Trumpet Signal with θ_k.

This plot was generated in Matlab using

```
T=linspace(0,1,44100);F=494;
C=[.1155 .3417 .1789 .1232 .0678];
C=[C .0473 .0260 .0045 .0020];
TH=[-2.13 1.67 -2.545 .661 -2.039];
TH=[TH 2.16 -1.0467 1.858 -2.39];
X=C*cos(2*pi*F*[1:9]'*T-TH'*ones(1,44100));
subplot(211),plot(T(1:200),X(1:200))
axis tight,grid on,soundsc(X,44100)
```

- The plots of the actual and synthetic trumpet signals are quite similar to each other;
- The actual and synthetic trumpet signals sound almost (but not exactly) the same;
- There is no DC (zero frequency) term. This makes sense: no one could hear it!
- The **timbre** (sound) of the trumpet is produced by the amplitudes $\{c_k\}$. The fundamental alone would be just a pure 494 Hertz tone; the **overtones** (harmonics) create the richer sound of the trumpet;
- The synthetic trumpet signal has no harmonics at 4940 Hertz or above, since these are very small in the actual trumpet signal;
- If the trumpet were playing a different note, 494 would be replaced by the appropriate frequency (in Hertz) given in the following table. In a different octave, all of these frequencies would be multiplied or divided by a power of two, depending on the octave.
- If we want to **store** the synthetic trumpet signal, we can store 44100 samples (numbers) for every second of trumpet sound, or we can store the 18 values of c_k and θ_k and **generate** the trumpet signal at any desired note by plugging in the frequency from the following table. Which method will allow you to store more tunes on your iPod?

Note:	A	A#	B	C	C#	D
Hertz:	440	466	494	523	554	587
Note:	D#	E	F	F#	G	G#
Hertz:	622	659	698	740	784	830

In fact, we may omit the phases θ_k in the synthetic trumpet signal without affecting its sound, although the plot of the signal looks quite different. Using $\theta_k = 0$ in the Fourier series gives

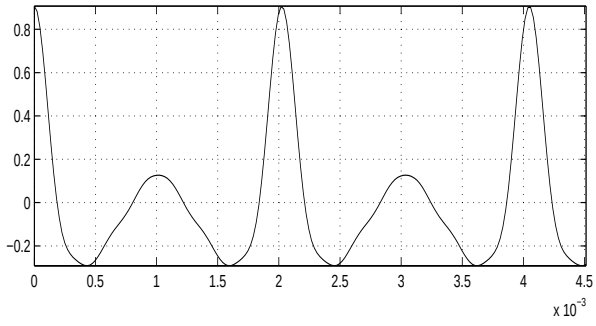


Figure 3.4: Synthetic Trumpet Signal: θ_k=0.

But the synthetic trumpet with θ_k=0 **sounds** just like the one with $\theta_k \neq 0$ (try it yourself).

3.2.3 Line Spectrum

Listing the c_k in a table is not a good way to specify the Fourier series. As noted in Chapter 2, humans think visually. So it makes sense to make a *bar graph* of the c_k against frequency $\frac{k}{P}$. This is called the **line spectrum** of the signal.

The line spectrum of the synthetic trumpet signal is in Figure 3.5. Note that the amplitudes c_k can be read off of the heights of the lines.

We will obtain this plot in Section 3.4.3. A separate **phase spectrum** plot of the θ_k vs. $\frac{k}{P}$ Hertz can also be made. However, since phase cannot be heard, we will omit phase spectrum plots in these notes.

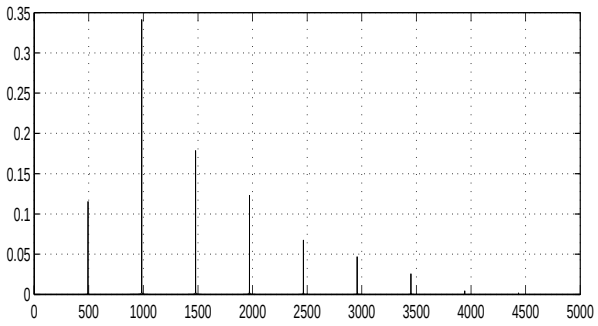


Figure 3.5: Synthetic Trumpet Line Spectrum.

3.2.4 Another Form of Fourier Series

The above depiction of the Fourier series uses cosines with phase shifts. Using the trig identity (1.5), we can rewrite this using sines and cosines:

$$\begin{aligned} x(t) = c_0 &+ \sum_{k=1}^{\infty} c_k \cos\left(2\pi\frac{k}{P}t - \theta\right) \\ x(t) = a_0 &+ \sum_{k=1}^{\infty} a_k \cos\left(2\pi\frac{k}{P}t\right) \\ &+ \sum_{k=1}^{\infty} b_k \sin\left(2\pi\frac{k}{P}t\right) \end{aligned} \tag{3.4}$$

where the coefficients are related by

$$\begin{aligned} a_k &= c_k \cos(\theta_k) \\ b_k &= c_k \sin(\theta_k) \\ c_k &= \sqrt{a_k^2 + b_k^2} \\ \theta_k &= \arctan[b_k/a_k] + 0 \text{ if } a_k > 0 \\ \theta_k &= \arctan[b_k/a_k] + \pi \text{ if } a_k < 0 \end{aligned} \tag{3.5}$$

3.3 Sampling Theorem

We prove this remarkable result for periodic signals, but the period can be arbitrarily large, so in practice it also applies to non-periodic signals.

This theorem was proved (using a different argument) by Claude Shannon (UM Class of 1936). A bust of his head is on the left side of the University of Michigan North Campus Diag entrance to the EECS Building.

3.3.1 Sampling Theorem: Derivation

- Let $x(t)$ be a real-valued continuous-time signal **bandlimited** to B Hertz. This means that its largest frequency is B Hertz;
- Let $x(\mathrm{t}=\frac{n}{S})$ be the sequence of numbers obtained by sampling $x(t)$ at a sampling rate of $S\frac{\mathrm{SAMPLE}}{\mathrm{SECOND}}$, that is, every $\frac{1}{S}$ seconds;
- Then $x(t)$ can be uniquely reconstructed from its samples $x(\frac{n}{S})$ if $\boxed{S > 2B}$

The minimum sampling rate $2B$ is called the Nyquist sampling rate. Although the actual units are $2B\frac{\mathrm{SAMPLE}}{\mathrm{SECOND}}$, this is usually abbreviated to $2B$ "Hertz," which has the same dimensions.

All of digital signal processing exists because of this theorem. It means that we can replace analog signal processing, which requires inductors, resistor and capacitors, with digital processing on computers and microprocessors.

Why is there any reason to think a signal can be reconstructed from its samples? Let $x(t)$ be a real-valued continuous-time signal that is

- Periodic with period P seconds;
- Bandlimited to $B = \frac{K}{P}$ Hertz;
- Sampled at $S = \frac{L}{P}\frac{\mathrm{SAMPLE}}{\mathrm{SECOND}}$.

Since $x(t)$ is periodic, it can be written as a sum of sinusoids at frequencies k/P for $k = 0, 1, 2 \ldots$. Since $x(t)$ is bandlimited, it has a maximum frequency which must be K/P for some integer K. So the Fourier series expansion of $x(t)$ is finite:

$$x(t) = a_0 + \sum_{k=1}^{K} \left[a_k \cos\left(2\pi \frac{kt}{P}\right) + b_k \sin\left(2\pi \frac{kt}{P}\right) \right] \tag{3.6}$$

So $x(t)$ is completely characterized by 2K+1 numbers $\{a_0, a_1 \ldots a_K, b_1 \ldots b_K\}$. Now if we can somehow obtain these 2K+1 numbers, we know $x(t)$ for all t, since we have a formula for $x(t)$. But how can we obtain them?

An idea: **sample** the signal! Sampling the signal at $L\frac{\mathrm{SAMPLE}}{\mathrm{PERIOD}}$ is the same as sampling it

$$L\frac{\mathrm{SAMPLE}}{\mathrm{PERIOD}} / P\frac{\mathrm{SECOND}}{\mathrm{PERIOD}} = \frac{L}{P}\frac{\mathrm{SAMPLE}}{\mathrm{SECOND}} = S\frac{\mathrm{SAMPLE}}{\mathrm{SECOND}}.$$

Setting $t = \frac{n}{S}$ where $n = 0, 1 \ldots L$–1 gives the L linear equations in 2K+1 unknowns

$$x\left(\frac{n}{S}\right) = a_0 + \sum_{k=1}^{K} \left[a_k \cos\left(2\pi \frac{kn}{PS}\right) + b_k \sin\left(2\pi \frac{kn}{PS}\right) \right]. \tag{3.7}$$

Note that PS is dimensionless. This system of linear equations has a unique solution if $L > 2K$:

$$L > 2K \rightarrow L/P > 2K/P \rightarrow S > 2B. \tag{3.8}$$

That is, the sampling rate must exceed the bandwidth. But note that the period P has cancelled, so this result is valid for any P. We can set $P = 10^{1000}$ seconds and the result still holds! This shows why exact recovery of a bandlimited signal from its samples is possible.

3.3.2 Sampling Theorem: Example

A periodic signal $x(t)$ with period=0.1 second is bandlimited to 20 Hertz. It is sampled at $50\frac{\mathrm{SAMPLE}}{\mathrm{SECOND}}$, i.e., every $\frac{1}{50}$=.02 seconds, resulting in the following samples:

t	.00	.02	.04	.06	.08
x	11.	4.014	4.739	3.115	-2.868

Since $x(t)$ has period=0.1 seconds, we have $x(.10) = x(.00)$, $x(.12) = x(.02)$, etc. The goal is to reconstruct $x(t)$ from its samples.

SOLUTION:

- Since the period=0.1 second, the harmonics have frequencies $\frac{k}{0.1}$=10k Hertz.
- Since the maximum frequency is 20 Hertz, there are only two ($K = 2$) harmonics.

The Fourier series expansion of $x(t)$ is

$$\begin{aligned} x(t) &= b_1 \sin(2\pi 10t) + b_2 \sin(2\pi 20t) \\ +a_0 &+ a_1 \cos(2\pi 10t) + a_2 \cos(2\pi 20t). \end{aligned} \quad (3.9)$$

Setting $t = \{.00, .02, .04, .06, .08\}$ gives five linear equations in five unknowns $\{a_0, a_1, a_2, b_1, b_2\}$:

$$\begin{aligned} x(.00) &= b_1 \sin(2\pi(.0)) + b_2 \sin(2\pi(.0)) \\ +a_0 &+ a_1 \cos(2\pi(.0)) + a_2 \cos(2\pi(.0)). \end{aligned}$$

$$\begin{aligned} x(.02) &= b_1 \sin(2\pi(.2)) + b_2 \sin(2\pi(.4)) \\ +a_0 &+ a_1 \cos(2\pi(.2)) + a_2 \cos(2\pi(.4)). \end{aligned}$$

$$\begin{aligned} x(.04) &= b_1 \sin(2\pi(.4)) + b_2 \sin(2\pi(.8)) \\ +a_0 &+ a_1 \cos(2\pi(.4)) + a_2 \cos(2\pi(.8)). \end{aligned}$$

$$\begin{aligned} x(.06) &= b_1 \sin(2\pi(.6)) + b_2 \sin(2\pi(1.2)) \\ +a_0 &+ a_1 \cos(2\pi(.6)) + a_2 \cos(2\pi(1.2)). \end{aligned}$$

$$\begin{aligned} x(.08) &= b_1 \sin(2\pi(.8)) + b_2 \sin(2\pi(1.6)) \\ +a_0 &+ a_1 \cos(2\pi(.8)) + a_2 \cos(2\pi(1.6)). \end{aligned}$$

Setting t=.10 gives the same equation as t=.00.

Inserting the samples of $x(t)$ and computing the various sines and cosines gives the five linear equations in five unknowns

$$11. = a_0 + a_1 + a_2. \quad (3.10)$$

$$\begin{aligned} 4.014 &= (.9511)b_1 + (.5878)b_2 \\ +a_0 &+ (.3090)a_1 - (.8090)a_2. \end{aligned} \quad (3.11)$$

$$\begin{aligned} 4.739 &= +(.5878)b_1 - (.9511)b_2 \\ +a_0 &+ -(.8090)a_1 + (0.3090)a_2. \end{aligned} \quad (3.12)$$

$$\begin{aligned} 3.115 &= -(.5878)b_1 + (.9511)b_2 \\ +a_0 &+ -(.8090)a_1 + (.3090)a_2. \end{aligned} \quad (3.13)$$

$$\begin{aligned} -2.868 &= -(.9511)b_1 - (.5878)b_2 \\ +a_0 &+ +(.3090)a_1 - (.8090)a_2. \end{aligned} \quad (3.14)$$

Note how the same four numbers keep appearing.

To solve five linear equations in five unknowns, we run the following Matlab program:

```
B=[11.,4.014,4.739,3.115,-2.868];
d=.3090;e=.5878;f=.8090;g=.9511;
A=[0 0 1 1 1;g e 1 d -f];
A=[A;e -g 1 -f d;-e g 1 -f d];
A=[A;-g -e 1 d -f];A\B'
```

The answer is `3,1,4,2,5` which are $\{b_1, b_2, a_0, a_1, a_2\}$ in that order. So the unique $x(t)$ satisfying all the conditions given above is:

$$\begin{aligned} x(t) &= 3\sin(2\pi 10t) + \sin(2\pi 20t) \\ +4 &+ 2\cos(2\pi 10t) + 5\cos(2\pi 20t). \end{aligned} \quad (3.15)$$

This was a lot of computation! Isn't there an easier way to do this? Read on...

3.4 Formulae for a_k and b_k

Even apart from the trouble of setting up the linear system of equations, solving them figures to be time-consuming. Fortunately, we can solve them in closed form (i.e., there are explicit formulae for the solution, into which we can plug).

3.4.1 Formulae for a_k and b_k

Let $x(t)$ have the following properties:

- Periodic with period=P seconds;
- Sampled at $S\frac{\text{SAMPLE}}{\text{SECOND}}$ so that
- $N = PS$ is an integer, resulting in
- Samples $x[n] = x(n/S)$ for $n = 0, 1 \ldots N-1$.
- Note $x[N]=x(\frac{N}{S})=x(\frac{PS}{S})=x(P)=x(0)=x[0]$ so $x[n]$ is periodic with period=N.

Then we have these formulae:

$$\begin{aligned} a_0 &= \frac{1}{N}\sum_{n=0}^{N-1} x[n] \\ a_k &= \frac{2}{N}\sum_{n=0}^{N-1} x[n]\cos(2\pi nk/N) \\ b_k &= \frac{2}{N}\sum_{n=0}^{N-1} x[n]\sin(2\pi nk/N). \end{aligned} \quad (3.16)$$

These formulae are derived in an Appendix of Lab #3. You are **not** responsible for this derivation! But do try working through it, with a cup of non-decaf coffee. It's not as bad as it looks.

Note that all the Fourier series coefficients $\{a_k, b_k, c_k\}$ are at frequency (in Hertz)

$$f = \frac{k}{P} = \frac{kS}{N}. \quad (3.17)$$

Sampling N times in a period P makes the sampling rate $S=\frac{N}{P}$, equivalent to $N=PS$.

3.4.2 Formulae for a_k and b_k: Example

Applying these to the example we just did gives

$$a_0 = \frac{1}{5}\sum_{n=0}^{4} x[n] = 4. \quad (3.18)$$

$$\begin{aligned} a_1 &= \frac{2}{5}\sum_{n=0}^{4} x[n]\cos(2\pi(1n)/5) &= 2. \\ a_2 &= \frac{2}{5}\sum_{n=0}^{4} x[n]\cos(2\pi(2n)/5) &= 5. \\ b_1 &= \frac{2}{5}\sum_{n=0}^{4} x[n]\sin(2\pi(1)n/5) &= 3. \\ a_1 &= \frac{2}{5}\sum_{n=0}^{4} x[n]\sin(2\pi(2)n/5) &= 1. \end{aligned}$$

That is certainly a lot easier than setting up and solving the previous linear system of equations!

An even easier way to compute the $\{a_k\}$ and $\{b_k\}$ is to use Matlab's `fft`, which computes

$$\texttt{fft(X)} = \sum_{n=0}^{N-1} x[n]e^{-2\pi i\frac{nk}{N}}, k = 0, 1 \ldots N-1. \quad (3.19)$$

because this quantity can be computed very quickly using the Fast Fourier Transform (FFT) algorithm. You will learn about the FFT if you take a DSP course in your junior or senior year.

To go from the output of `fft` to $\{a_k\}$ and $\{b_k\}$ requires some attention. Proceed as follows:

- Form `X`=[x[0],x[1]...x[N–1]].
- Compute >>`F=fft(X);` in Matlab.
- a_0=`F(1)/N`
- a_k=`2*real(F(k+1))/N` for $k \geq 1$.
- b_k=`-2*imag(F(k+1))/N`
- c_k=`2*abs(F(k+1))/N`
- θ_k=`angle(F(k+1))`

For the example we just did, use:

```
X=[11.  4.014 4.739 3.115 -2.868];
```

```
F=fft(X);A=2*real(F)/5;
B=-2*imag(F)/5;C=2*abs(F)/5;
```

The result of this program is:

```
A=[8 2 5 5 2]
B=[0 3 1 -1 -3]
C=[8 3.6 5.1 5.1 3.6]
```

I used this to get the c_k and θ_k values in the table for the trumpet signal. Note the following:

- `A(1)` is **double** a_0, so a_0=$\frac{8}{2}$=4.
- `A(2)` and `A(3)` are a_1=2 and a_2=5.
- `B(1)` is zero, as it should (b_0=0 always).
- `B(2)` and `B(3)` are b_1=3 and b_2=1.
- The second halves of `A,B,C` are the mirror images of the first halves of `A,B,C`.
- **IGNORE THE SECOND HALF OF THE OUTPUT OF `fft`!**
- For music signals, $a_0 = 0$ anyway, so don't worry about `A(1)` being double a_0.

Note there is an issue of **indexing**:

- Indexing of $\{a_k, b_k, c_k\}$ starts at k=0.
- Indexing of Matlab's `A,B,C` starts at k=1.
- So a_k=`2*real(F(k+1))/N`, etc.
- To compute the frequency f corresponding to a peak in `abs(fft(X))` at Matlab index `K`, use $\boxed{f = (\mathtt{K}-1)\frac{S}{N}}$.

3.4.3 Computing Line Spectra

To compute the line spectrum of a signal using `fft`, proceed as follows:

The signal has been sampled at $S\frac{\text{SAMPLE}}{\text{SECOND}}$, resulting in samples stored in Matlab vector `X`=$\{x[n], n = 0 \ldots N-1\}$. Now do this:

```
N=length(X);F=[0:N-1]*S/N;
FX=2*abs(fft(X))/N;
plot(F(1:N/2),FX(1:N/2))
```

I used this to plot the line spectrum of the synthetic trumpet signal shown in Figure 3.5:

```
%Generate synthetic trumpet:
T=linspace(0,1,44100);F=494;
C=[.1155 .3417 .1789 .1232 .0678];
C=[C .0473 .0260 .0045 .0020];
X=C*cos(2*pi*F*[1:9]'*T);
%Compute its line spectrum:
F=[0:43999]*44100/44100;
FX=2*abs(fft(X))/44100;
plot(F(1:5000),FX(1:5000))
```

3.5 Noisy Periodic Signals

Consider the noisy trumpet signal formed by adding **noise** to the synthetic trumpet signal:

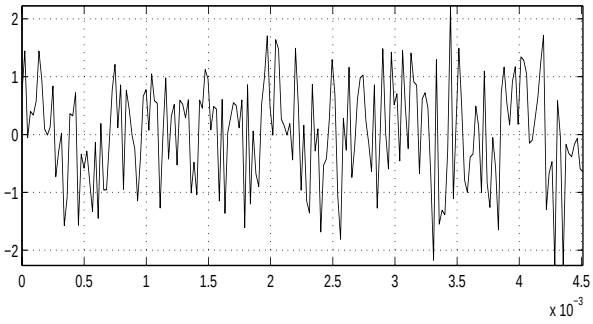


Figure 3.6: Noisy Synthetic Trumpet Signal.

It is hard to tell from the plot that there is anything but noise! Listen to the signal: the trumpet is only faintly audible over the noise.

Now look at the spectrum of the noisy signal:

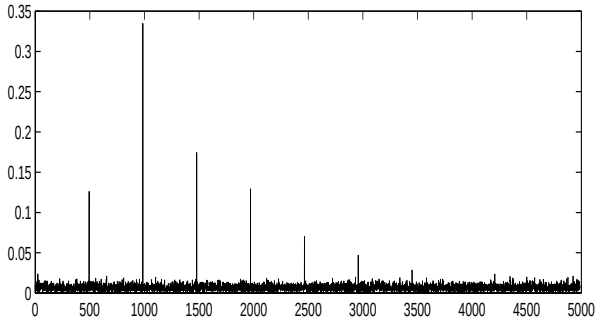


Figure 3.7: Line Spectrum of Noisy Trumpet.

The trumpet harmonics stick up over the noise spectrum like dandelions sticking up over a grassy lawn. It is clear that we can eliminate most of the noise by simply setting all of the spectrum values that are not trumpet signal harmonics to zero. An easy way to do this is to threshold the spectrum values by setting them to zero unless they are large; see Figure 3.8.

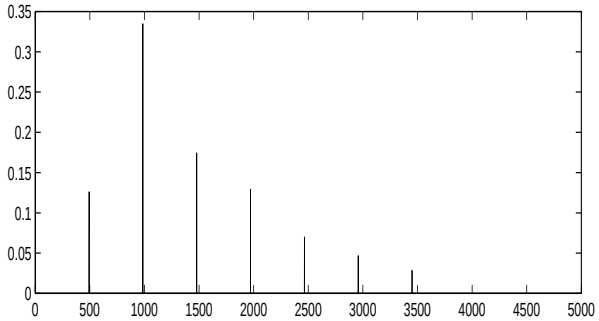


Figure 3.8: Cleaned-Up Line Spectrum.

Note we have not completely eliminated the noise, since harmonics themselves have noise added to them. Also, the smallest two harmonics were thresholded to zero, since they are smaller than most of the noise and are now drowned in noise. There is nothing we can do about that.

But most of the noise has been eliminated, using the fact that the synthetic trumpet spectrum is zero except at the frequencies kP Hertz.

We could read off the c_k and θ_k from the output of `fft` and plug them into the Fourier series. However, it is easier to use Matlab's `ifft`, as shown below. The result is Figure 3.9:

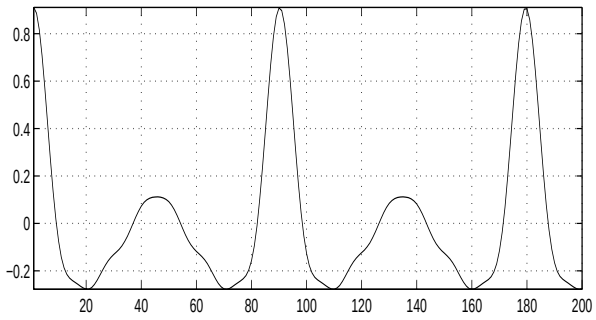


Figure 3.9: Cleaned-Up Noisy Trumpet.

We have eliminated most of the noise. And listening to the cleaned-up signal, almost all of the noise is gone! This is a simple example of **filtering** a noisy signal: We allow only certain frequencies to go through the filter, and we eliminate (filter out) all of the other frequencies.

These plots were all generated using this:

```
%Generate synthetic trumpet:
T=linspace(0,1,44100);F=494;
C=[.1155 .3417 .1789 .1232 .0678];
C=[C .0473 .0260 .0045 .0020];
X=C*cos(2*pi*F*[1:9]'*T);
%Add noise to it:
Y=X+0.8*randn(1,44100);
%Compute spectra:
```

```
F=[0:43999]*44100/44100;
FY=2*abs(fft(Y))/44100;
figure,plot(T(1:200),Y(1:200))
figure,plot(F(1:5000),FY(1:5000))
%Threshold noisy spectrum:
%Use 0.025 since it works.
FZ=FY;FZ(abs(FZ)<0.025)=0;
Z=44100/2*real(ifft(FZ));
figure,plot(T(1:200),Z(1:200))
figure,plot(F(1:5000),FZ(1:5000))
```

3.6 Low-Pass Filtering

3.6.1 Basic Concept of Filtering

A more general form of filtering to reduce noise is low-pass filtering. The idea behind low-pass filtering is that many real-world signals consist primarily of low frequencies, while noise consists of low and high frequencies. So if we simply eliminate the high frequencies in a noisy signal, we will be eliminating mostly noise. Keeping or passing the low frequencies will preserve most of the signal, while also keeping some of the noise.

Low-pass filtering shows where the term "filtering" came from: passing small particles (frequencies) while stopping large particles (frequencies) is what a paper filter does. It does not work as well as filtering periodic signals, since now we know much less about the signal, and so we need to pass all of the low frequencies, not just the frequencies of the individual harmonics.

The following example shows how to eliminate frequencies above a *cutoff frequency* F Hertz:

3.6.2 Low-Pass Filtering Example

- A one-Hertz sinusoid has noise added to it.
- It is sampled at $S = 1000\frac{\text{SAMPLE}}{\text{SECOND}}$.
- The snippet of noisy signal is 1 second long.
- Use a low-pass filter with cutoff F=2 Hertz.

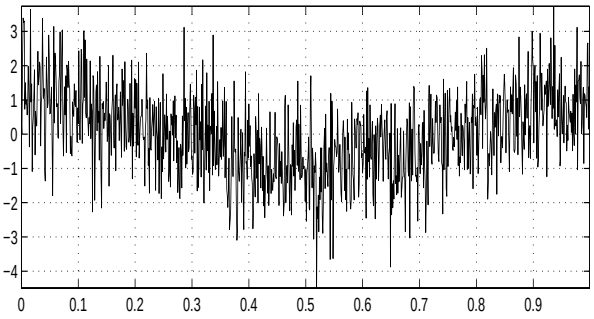


Figure 3.10: Noisy Sinusoidal Signal.

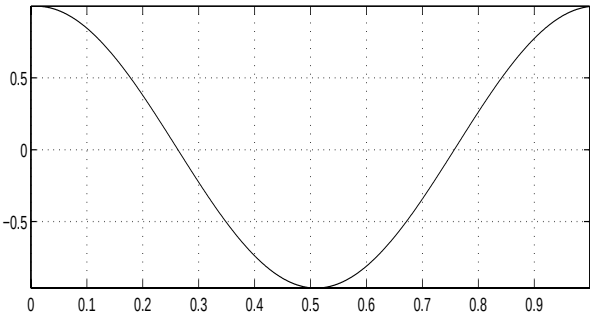


Figure 3.11: Low-Pass Filtered Sinusoid.

```
T=linspace(0,0.999,1000);
Y=cos(2*pi*T)+randn(1,1000);F=2;
S=1000;N=length(Y);K=1+N*F/S;
plot(T,Y) %From F=(K-1)S/N
FY=fft(Y);FY(K:1002-K)=0;
Z=real(ifft(FY));plot(T,Z)
```

This is an extreme example; we filter all frequencies above one very-low-frequency sinusoid. But the results are quite dramatic!

Chapter 4

Spectrogram: Time-Varying Spectra

4.1 Spectrogram: Overview

In Chapter 3 we assumed that the signal $x(t)$ was periodic, so that $x(t) = x(t+P)$ for some P and for **all** time t. That is:

- A signal is periodic only if it is periodic for **all** time $-\infty < t < \infty$.
- A signal is sinusoidal only if it is a sinusoid for **all** time $-\infty < t < \infty$.

Of course, no signal, musical or otherwise, can be known to have these properties (how can we know what will happen in the infinite future?).

Musical signals (an instrument playing music) actually have the form

$$\begin{aligned} x(t) &= \sum_{k=1}^{\infty} A_{k1} \cos\left(2\pi \frac{k}{P_1} t + \theta_1\right), T_0 < t < T_1 \\ x(t) &= \sum_{k=1}^{\infty} A_{k2} \cos\left(2\pi \frac{k}{P_2} t + \theta_2\right), T_1 < t < T_2 \\ x(t) &= \sum_{k=1}^{\infty} A_{k3} \cos\left(2\pi \frac{k}{P_3} t + \theta_3\right), T_2 < t < T_3 \\ &\vdots \quad \vdots \quad \vdots \end{aligned} \tag{4.1}$$

That is, the line spectrum of $x(t)$ **changes** every so often. The study of signals whose spectra changes with time is time-frequency analysis.

Fortunately, for music signal processing, we have three huge advantages:

1. The **durations** T_{i+1}–T_i are known. All whole notes have the same duration; all half notes have half the duration of whole notes, etc. So we do not have the problem of **segmenting** the signal into different intervals;
2. The **frequencies** $\frac{k}{P_i}$ can only take on twelve different values in an octave. So we are only **choosing** from several possible frequencies;
3. The **phases** can be assumed to be zero for both musical synthesis and for the musical transcription from the synthesizer output.

This section introduces the lowest-level form of the **spectrogram**, which is a way of visualizing the time-varying line spectrum of a signal. The spectrogram, like log-log and semi-log plots, is a means of interpreting data.

4.2 Spectrogram: Motivation

For example, we are given the signal in the file `victorstone.mat`. All we know about it is that it was sampled at $8192 \frac{\text{SAMPLE}}{\text{SECOND}}$. We wish to intepret this signal-what's going on in it?

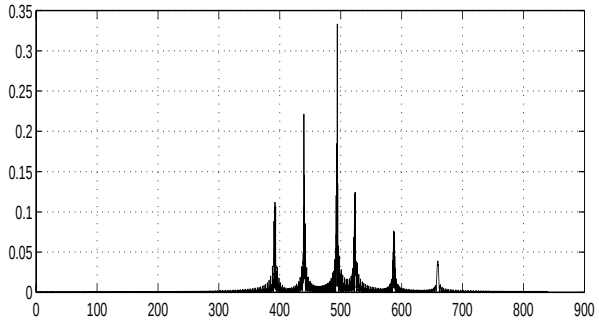


Figure 4.1: Line Spectrum of Unknown Signal.

Fig. 4.1 is its line spectrum-not much help! This plot was generated in Matlab using

```
load victorstone.mat
N=length(X);S=8192;
FX=2/N*abs(fft(X));
F=[0:N-1]*S/N;
plot(F(1:8000),FX(1:8000))
```

4.3 Spectrogram: Example

4.3.1 Spectrogram: Presentation

Fig. 4.2 is its spectrogram-more useful!

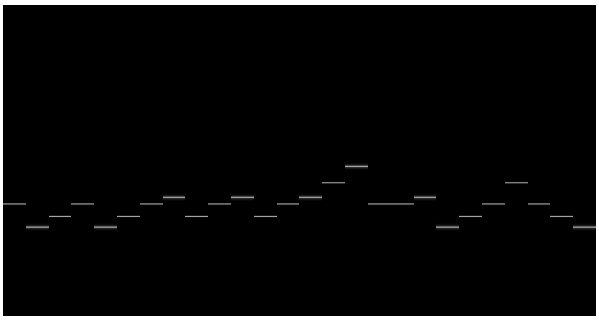


Figure 4.2: Spectrogram of Unknown Signal.

This plot was generated in Matlab using

```
load victorstone.mat
LX=length(X);L=26;N=LX/L;
XX=reshape(X',N,L);
FXX=abs(fft(XX));
FXX=FXX(5*N/6:N,:);
subplot(211),imagesc(FXX)
colormap(gray),axis off
```

4.3.2 Spectrogram: Discussion

- The **height** of each line is the frequency kP_i in Hertz of that time segment of the signal;
- The **length** of each line is the duration T_{i+1}–T_i in seconds of that time segment;
- The **brightness** of each line is the amplitude A_{ki} of that time segment of the signal.

The line heights are at the frequency values used by musical notes. The notes change as time progresses from left to right. You can now recognize this as "The Victors." Indeed, the spectrogram can function as a crude type of musical notation, indicating the pitch and duration of notes played in succession. It could also function like a player piano roll, moving from right to left.

Details of the spectrogram for this signal:

- `X` had length `LX`=78000. At $8192\frac{\text{SAMPLE}}{\text{SECOND}}$ this is a duration of $\frac{78000}{8192}$=9.5215 seconds.
- `X` was segmented into L=26 segments of length N=3000 samples each. This is a duration of $\frac{3000}{8192}$=0.3662 seconds.
- `X` was laid out by column in the array `XX`:
  - `X(1:3000)` in the 1^{st} column of `XX`;
  - `X(3001:6000)` in the 2^{nd} column of `XX`;
  - `X(75001:78000)` in the 26^{th} column.

- `fft` when applied to an array computes the FFT of each column of the array. So `FXX` is the array of line spectra of each segment of `X`, only laid out vertically, rather than horizontally.
- Only the bottom sixth of FXX is kept. The top half is the mirror image of the lower half, so it should not be shown.
- `imagesc` displays this array as an image. The brightness of each image pixel is `FXX` at that frequency and time.

If you really want to be lazy, you can plot the spectrogram of a signal consisting of `L` segments of lengths `N` each, where `N=length(X)/L`, using

```
imagesc(abs(fft(reshape(X',N,L))))
```

To get a 3-D plot of the spectrogram, use

```
waterfall(abs(fft(reshape(X',N,L))))
```

This is snazzy-looking, but hard to interpret. Matlab's signal processing toolbox (present on all CAEN machines, but not on some Central Campus computer labs) has the command `specgram`, which allows you to do many things beyond the scope of this course.

How did we know to segment `X` into 26 segments? If we know the signal is a musical signal consisting of whole notes with durations 0.3662 seconds, we know that the number of notes is $\frac{9.5215}{0.3662}$=26 since the duration of the signal is 9.5215 seconds.

4.4 Time-Frequency Plots: Resolution Tradeoff

We don't have to know how many segments the signal actually contains. In fact, varying

- The number of segments `L` and
- The length `N` of each segment so
- `L*N=LX=`total length of the signal.

trades off time and frequency resolution. The larger `L` is, the shorter the length `N` of each segment, so changes in the spectrum as it changes in time can be tracked faster. However, `fft` computes line spectra at frequencies f at Matlab indices `K`, where

$$f = \frac{K-1}{P} = (K-1)\frac{S}{N} \tag{4.2}$$

$P = \frac{N}{S}$ is now the duration of each interval. So the discretization, hence the resolution, of frequency is coarser.

To illustrate this, recompute the above spectrogram using different values of `L` and hence `N`:

1. `L`=13 and `N=length(X)/L`=6000;
2. `L`=104 and `N=length(X)/L`=750.

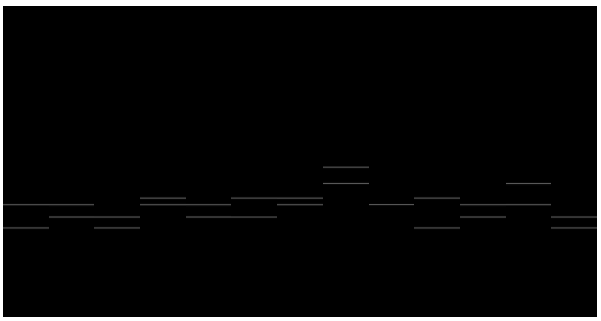


Figure 4.3: Spectrogram with L=13 segments.

Each segment now contains two notes, and the spectrogram plots both of them. If you were playing "The Victors" from this, you would have to guess which note to play in each segment!

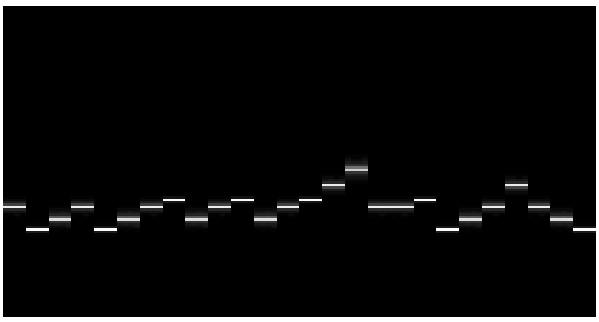


Figure 4.4: Spectrogram with L=104 segments.

Each segment is only 750 samples long, and the frequencies are smeared out. This improves visibility, but is not so good for actually computing the frequencies.

4.5 Chirp Signal

A common test signal for time-frequency analysis is the **chirp**, which does indeed sound like a bird chirp (dolphin clicks are also chirps).

$$x(t) = A\cos(2\pi F t^2) \text{ for } t > 0. \tag{4.3}$$

The chirp $\cos(t^2)$ is plotted in Figure 4.5:

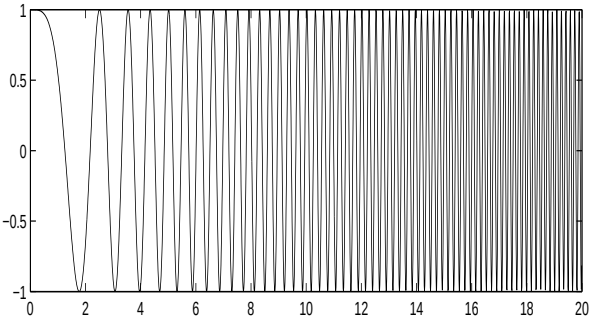


Figure 4.5: Chirp Signal $\cos(t^2)$.

The chirp signal looks like a sinusoid whose frequency is steadily increasing in time.

Its spectrogram is plotted in Figure 4.6:

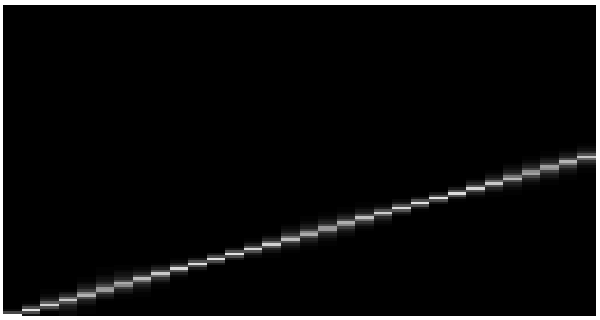


Figure 4.6: Spectrogram of Chirp.

This clearly indicates a signal whose frequency is increasing *linearly* with time. The spectrogram makes interpretation of the signal easy.

These plots were generated in Matlab using

```
X=cos([0:8191].*[0:8191]/10000);
T=linspace(0,19.99,2000);
subplot(211),plot(T,X(1:2000))
FXX=abs(fft(reshape(X',256,32)));
FXX=FXX(129:256,:);
figure,subplot(211),imagesc(FXX)
axis off,colormap(gray)
```

4.5.1 Interpretation of Chirp

How do we interpret exactly what is going on in the spectrogram of the chirp?

- $\cos(t^2)$ is sampled at $S = 100\frac{\text{SAMPLE}}{\text{SECOND}}$. Setting $t = n/100$ in $\cos(t^2)$ gives the samples $x[n] = \cos(n^2/1000)$, stored in `X`.

- The length of `X` is 8192 samples, so its duration is $\frac{8192}{100}$=81.92 sec (almost 1.5 minutes!).

- The height of the right-most line in the spectrogram is index K=67. This is frequency $f = (K-1)\frac{S}{N} = (67-1)\frac{100}{256} = 25.8$ Hertz since each segment is 256 samples long.

- The *instantaneous frequency* of $\cos(2\pi f t^2)$ is $2ft$, not just ft as you might expect from writing $\cos(2\pi f t^2)$=$\cos(2\pi(ft)t)$. Here, $f = \frac{1}{2\pi}$, so the instantaneous frequency at $t = 81.92$ is $\frac{2(81.92)}{2\pi}$=26.1 Hertz. This is slightly higher than the spectrogram value since the spectrogram averages over the final segment, instead of using the value at its end.

4.6 Removing Interference

To show how a spectrogram can help in removing an interfering signal, **listen** to the signal:

```
load victorstone.mat
LX=length(X);S=8192;
X=X+cos(2*pi*700*[1:LX]/S);
soundsc(X,S)
```

Plot its spectrum and spectrogram using:

```
F=[0:LX-1]*S/LX;
FX=2/LX*abs(fft(X));
plot(F(1:9000),FX(1:9000))
%Now plot spectrogram:
LX=length(X);L=26;N=LX/L;
XX=reshape(X',N,L);
FXX=abs(fft(XX));
FXX=FXX(5*N/6:N,:);
subplot(211),imagesc(FXX)
colormap(gray),axis off
```

The spectrum is plotted in Figure 4.7.
The spectrogram is plotted in Figure 4.8.

The spectrum and spectrogram *together* show that the interference is a tone at 700 Hertz.

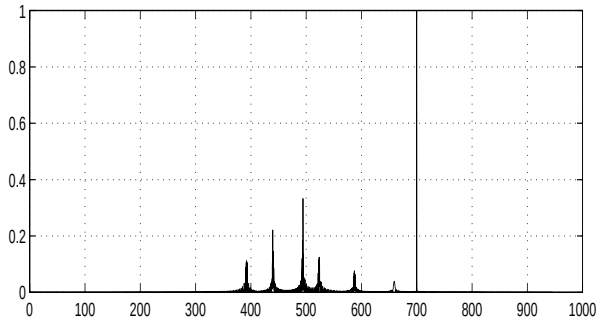


Figure 4.7: Spectrum: Victors+Interference.

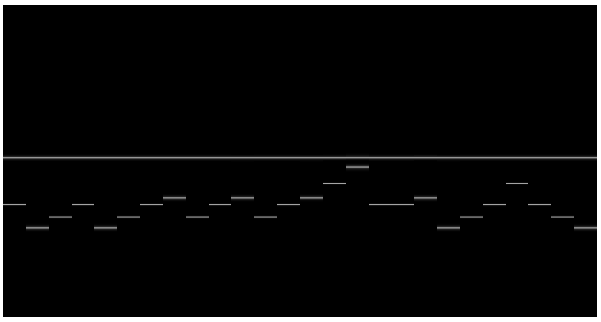


Figure 4.8: Spectrogram: Victors+Interference.

To eliminate this interference, run this:

```
K=1+round(LX*700/S);
KK=[K-100:K+100];FX=fft(X);
FX(KK)=0;FX(LX+2-KK)=0;
Y=real(ifft(FX));soundsc(Y,S)
```

- `K=1+round(LX*700/S);` comes from solving `f=(K-1)S/LX` for `K`;

- We set not only `FX(K)` to zero, but also all `FX` values whose indices are within 100 of `K`. We also set to zero the mirror image values;

- Listen to `Y`: The interference is gone!

In Lab #3 you will use the spectrogram to remove noxious interference from "The Victors."

Chapter 5

Transcriber Approaches

5.1 Overview

This chapter presents four procedures for identifying the note played by a musical instrument, a major part of the main project transcriber:

- **Spectrogram:** We have already seen that the spectrogram of an instrument playing several notes in succession (i.e., of music) depicts the frequencies and durations of the notes, and so functions as a crude type of musical notation. To map the spectrogram to musical staff notation is not difficult.
- **Fundamental:** The simplest way of identifying a musical note is to compute its FFT and see at which of 392,414,440 etc. Hertz a large peak (the fundamental) is present.
- **Harmonic Product Spectrum (HPS):** Downsampling (omitting some frequencies) the line spectrum of the note (computed using the FFT) and multiplying the resulting spectra emphasizes the fundamental.
- **Autocorrelation:** The *autocorrelation* (defined below) of a periodic signal is itself periodic, with the same period as the original signal. The time lag at which the autocorrelation peaks is the period.

This chapter also presents three other concepts pertinent to the project:

- **Additive synthesis** creates a synthetic musical instrument by creating and summing harmonics. The synthetic trumpet signal was generated using additive synthesis. Your team will create its own instrument.
- **Reverb(eration)** makes any sound seem richer by adding slightly delayed copies of itself. You will use this to make the single trumpet sound like the trumpet section of the University of Michigan marching band.
- **Signal-to-Noise Ratio (SNR)** is a measure of noise strength relative to signal strength. You will use this to evaluate the performance of your transcriber.

5.2 Note Identification

We present four methods for identifying a musical note played by an instrument. All of these methods have been used for musical transcription. In each term of Engineering 100, three of the four has been used by at least one team. Your team will choose one method, and will be

required to justify your choice (there is no single right answer; what works best for you?).

Demo codes that demonstrates each method. They are **not** sufficient for your transcriber!

5.2.1 Spectrogram

The spectrogram is an obvious choice for identifying the notes played by a single instrument. The spectrogram of an actual solo trumpet playing "The Victors" is shown in Figure 5.1:

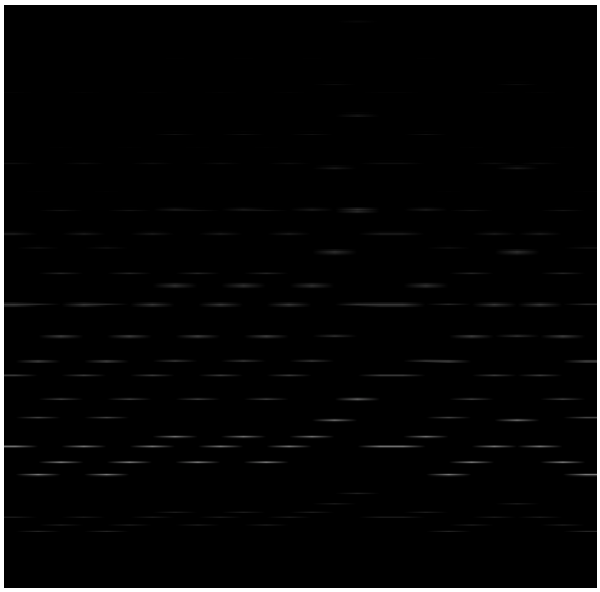


Figure 5.1: Spectrogram of Solo Trumpet Playing "The Victors."

The Matlab code used to generate this plot:

```
load proj2.mat
L=26;N=length(Y)/L;
YY=reshape(Y',N,L);
FYY=abs(fft(YY));
FYY=FYY(N-2999:N,:);
imagesc(FYY),colormap(gray),axis off
```

Note the following about this spectrogram:

- I used the synthesizer for the main project to play "The Victors" on the trumpet. `proj2.mat` is the synthesizer output file; it contains variable `Y`;
- The lowest 3000 rows are shown;
- Look at each column separately. The harmonics of each note are equally spaced in the vertical direction (frequency). They get smaller with increasing frequency (upward);
- For the trumpet, the second harmonic has roughly triple the amplitude of the fundamental. Some of the fundamentals are faint;
- Some harmonics of different notes are almost identical. For example, the fourth harmonic of G and third harmonic of C are at $4(392) = 1568 \approx 1569 = 3(523)$ Hertz.

The advantages of using the spectrogram are:

- Fundamentals and harmonics are apparent;
- Both frequencies and durations can be read.

The disadvantages of using the spectrogram are:

- The spectrogram has considerable clutter;
- A huge amount of computation is required.

5.2.2 Fundamental Identification

We have seen that the line spectrum of an instrument playing a single note consists of a fundamental frequency, and harmonic frequencies that are integer multiples of the fundamental frequency. Also, the fundamental frequency is one of 392, 414, 440 etc. Hertz, so *we only need to choose which frequency.* This suggests that we

compute the line spectrum at only those frequencies, and see at which one the line spectrum is large. In fact, the FFT is so fast that it is almost as fast to compute the entire line spectrum as just its values at 13 frequencies.

The line spectrum of an actual trumpet note:

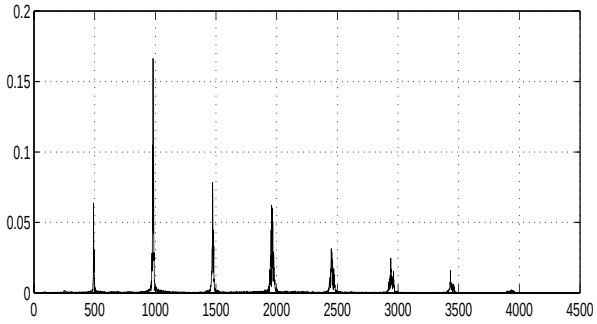


Figure 5.2: Line Spectrum of Actual Trumpet.

The Matlab code used to generate this plot, and to identify the note from the largest peak is:

```
load trumpet.mat;LX=length(X);
FX=2*abs(fft(X))/LX;
F=[0:2999]*44100/LX;
subplot(211),plot(F,FX(1:3000));
KMIN=round(LX/44100*370);
KMAX=round(LX/44100*850);
[G,K]=max(FX(KMIN:KMAX));
f=(K+KMIN-2)*44100/LX;
```

The program looks for the largest value of FX between frequencies 370 and 850 Hertz (Matlab indices `KMIN` and `KMAX`). This occurs at Matlab index `K+KMIN-1`. from which the frequency in Hertz of the peak is computed to be 491 Hertz.

The advantages of using the fundamental are:

- It is very fast, computationally;
- It works very well in additive noise.

The disadvantage of using the fundamental is that it may mistake the first harmonic for the fundamental. This will definitely happen for the trumpet playing the low G (392 Hertz), since its first harmonic is 2(392)=784 Hertz, which is also the fundamental for the high G (784 Hertz). So the low G will be identified as a high G!

This is an example of the **octave** problem, which is a well-known problem in music transcription. I leave it to you to figure it out.

5.2.3 Harmonic Product Spectrum

1. Compute spectrum `FX=abs(fft(X));`
   FX has peaks at 494,2(494),3(494)... Hertz.
2. Delete every other value of the spectrum:
   `FX2=FX(1:2:length(FX));`
   FX2 has peaks at $\frac{1}{2}$(494),494,$\frac{3}{2}$(494)...
3. Delete 2 out of every 3 spectrum values:
   `FX3=FX(1:3:length(FX));`
   FX3 has peaks at $\frac{1}{3}$(494),$\frac{2}{3}$(494),494...
4. Multiply these: `FXH=FX.*FX2.*FX3.;` Then `FXH` will have a huge peak at 494 Hertz.

`FXH` is the Harmonic Product Spectrum (HPS).

The HPS of an actual trumpet playing note B:

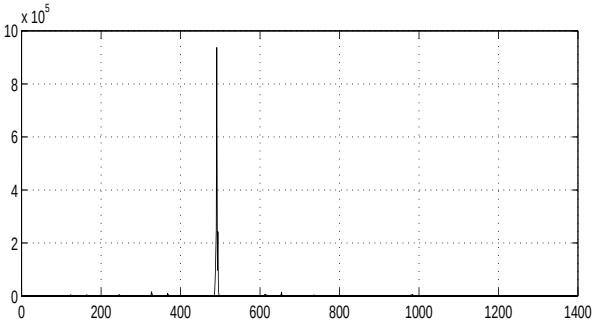


Figure 5.3: HPS of Actual Trumpet.

There is a single huge peak at 494 Hertz.

The Matlab code used to generate this plot (4, not 3, harmonic spectra are used here):

```
load trumpet.mat;LX=length(X);
FX=abs(fft(X));FX2=FX(1:2:LX);
FX3=FX(1:3:LX);FX4=FX(1:4:LX);
L4=length(FX4);
FH=FX(1:L4).*FX2(1:L4);
FH=FX3(1:L4).*FX4(1:L4);
F=[0:999]*44100/LX;
subplot(211),plot(F,FH(1:1000))
```

The advantages of using HPS are:

- It eliminates the octave problem if enough `FX`'s are used;
- It is very fast, computationally;
- It works very well in additive noise.

The disadvantages of using HPS are:

- If one harmonic is small, then `FXH` is also (HPS is vulnerable to small harmonics);
- It doesn't work at all for pure tones, since pure tones have no harmonics at all!

5.2.4 Autocorrelation

Unlike the other three methods, autocorrelation doesn't use the concept of line spectrum at all. It computes the period P of a periodic signal.

The *autocorrelation* $r[n]$ of $x[n]$ is defined as

$$r[n] = \sum x[i]x[i+n] = \sum x[i]x[i-n]. \quad (5.1)$$

In particular, some autocorrelation values are

$$\begin{aligned} r[0] &= x[0]x[0] + x[1]x[1] + x[2]x[2] \ldots \\ r[1] &= x[0]x[1] + x[1]x[2] + x[2]x[3] \ldots \\ r[2] &= x[0]x[2] + x[1]x[3] + x[2]x[4] \ldots \\ r[3] &= x[0]x[3] + x[1]x[4] + x[2]x[5] \ldots \end{aligned} \quad (5.2)$$

The idea behind using autocorrelation is:

- $r[0] = \sum x[i]^2$ is clearly *very* large;
- $r[n] = \sum x[i]x[i+n]$ is the sum of random products, and so it is relatively small;
- But if $x[n]$ is periodic with period=N then:
- $r[N] = \sum x[i]x[i+N] = \sum x[i]x[i] = r[0]$ is large, since $x[i+N] = x[i]$.
- So $r[n]$ has large peaks at $n = 0, N, 2N \ldots$

So to use autocorrelation to find the period:

1. The sampling rate used is $S\frac{\text{SAMPLE}}{\text{SECOND}}$.
2. Compute $r[n]$ from $x[n]$.
   A fast algorithm for computing $r[n]$ is
   `(ifft(abs(fft(X,2*length(X)).^2))`
3. Compute $\tilde{r}[n] = \frac{r[n]}{r[0]}$. Then $\tilde{r}[0]$=1;
4. Find the smallest $N \neq 0$ so $\tilde{r}[N] \approx 1$;
5. $P = \frac{N}{S}$ is the period in seconds.

The autocorrelation of an actual trumpet playing note B is shown in Figure 5.4.

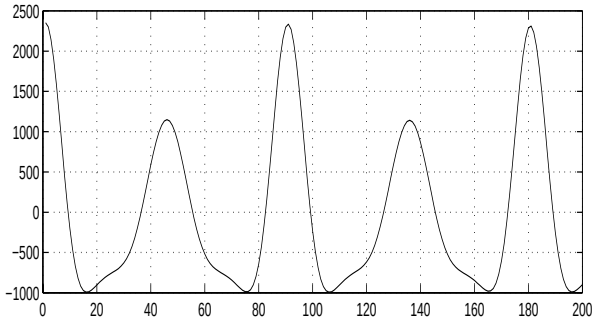


Figure 5.4: Autocorrelation of Actual Trumpet.

The Matlab code used to generate this plot:

```
load trumpet.mat;LX=length(X);
R=real(ifft(abs(fft(X,2*LX)).^2));
subplot(211),plot(R(1:200))
```

$r[0]$ is 2352, and $r[90]$ is slightly less at 2336. The sampling rate is $44100\frac{\text{SAMPLE}}{\text{SECOND}}$. The period is $P=\frac{91-1}{44100}$, and the fundamental frequency is $f=\frac{1}{P}=\frac{44100}{91-1}$=490 Hertz.

Note the peaks occur at Matlab indices 1 and 91, so the interval is 90. This should not be confused with f=(K–1)$\frac{S}{N}$: we are not using the line spectrum here! It's Matlab indexing again.

Compare 490 Hertz with the 491 Hertz obtained from fundamental frequency identification. There is a discretization issue in the exact peak location, so they don't agree exactly. And the trumpet seems to be slightly out of tune.

The peak at $r[90]$ (2336) is slightly less than $r[0]$ (2352), and the peak at $r[180]$ is slightly smaller still (2312). The reason this happens is that $x[n]$ has finite length, and as n gets larger $r[n]$ is computing by summing fewer terms. This could be corrected by dividing each $r[n]$ by the number of terms being summed, but it is hardly necessary if we are only interested in the first autocorrelation peak away from $n = 0$.

The advantages of using autocorrelation are:

- There are no fundamental vs. harmonics issues, unlike fundamental identification, so the octave problem is avoided;
- Zero harmonics no problem, unlike HPS;
- It works well in noise, since the noise is concentrated in $r[0]$. In fact, the autocorrelation $r[n]$ of zero-mean white Gaussian noise is large at $n = 0$ and very small elsewhere;
- Since $r[N] < r[0]$ if there is noise added, use the location of the peak value of $r[n]$ away from $n = 0$, not where $r[N] = r[0]$.

5.2.5 Autocorrelation with Noise

To expand on the last point, now suppose that zero-mean *white noise* $w[n]$ has been added to $x[n]$. Zero-mean white noise is a common model for noise. $w[n]$ is white noise means that:

- $w[i]$ and $w[j]$ are statistically uncorrelated with each other. This means that knowing $w[i]$ tells you little about the value of $w[j]$, even when i–j is small (of course we do need $i \neq j$ here).
- The autocorrelation of $w[n]$ is very large for $n = 0$, and very small otherwise.
- The mean $\frac{1}{M}\sum_{n=1}^{M} w[n]$ of $w[n]$ is roughly zero for large M.

Suppose we observe not $x[n]$ but

$$y[n] = x[n] + w[n]. \tag{5.3}$$

The autocorrelation $r_y[n]$ of $y[n]$ is

$$\begin{aligned} r_y[n] &= \sum y[i]y[i+n] \\ &= \sum (x[i]+w[i])(x[i+n]w[i+n]) \\ &= \sum x[i]x[i+n] + \sum w[i]w[i+n] \\ &= \sum w[i]x[i+n] + \sum x[i]w[i+n] \\ &\approx r_x[n] + r_w[n] \approx r_x[n] + \sigma^2\delta[n] \end{aligned} \tag{5.4}$$

since

$$\begin{aligned} \sum w[i]x[i+n] &\approx 0 \\ \sum x[i]w[i+n] &\approx 0 \\ \sum w[i]w[i+n] &\approx \sigma^2\delta[n] \end{aligned} \tag{5.5}$$

where

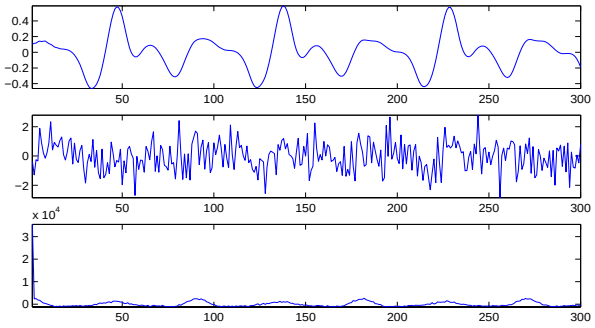
$$\begin{aligned} \delta[n] &= 1 \text{if } n = 0 \\ \delta[n] &= 0 \text{if } n \neq 0 \\ \sigma^2 &= \text{energy of } w[n]. \end{aligned} \tag{5.6}$$

More properly, $\frac{1}{M}\sum_{n=1}^{M} w[n]^2$ is the power of $w[n]$ for large M.

This analysis shows that the white noise added to $x[n]$ manifests itself as an increase in $r_y[0]$ from $r_x[0]$ to $r_x[0]+\sigma^2$. This means that the value of $r_y[0]$ cannot be used as a guide for the value of $r_x[N]$. Often, σ^2 is known approximately, so it can be subtracted from $r_y[0]$.

The following three plots shows how autocorrelation works on noisy signals. The plots are:

1. Noiseless trumpet signal;
2. Trumpet signal with noise added;
3. Autocorrelation of the noisy trumpet signal.



The only effect of the added noise on the autocorrelation is its larger value at $n = 0$. So the period of the noisy trumpet signal can still be determined to be 90 using autocorrelation. The previous procedure must be modified since now $r_y[0] > r_x[N] = r_x[2N] = \ldots$. But try estimating the period by examining the second plot of the noisy trumpet signal!

5.2.6 Searching for Known Periods

In Engin 100, we have important additional information: The fundamental frequencies of musical notes are known. Instead of attempting to estimate the period N, we are attempting to *choose* among a finite set of known values $\{N_1, N_2 \ldots N_{13}\}$ of N. So we need only see which of $\{r_y(N_1), r_y(N_2) \ldots r_y(N_{13})\}$ is largest.

To illustrate this, suppose we have noisy observations $y[n]$ of a signal sampled at $1200\frac{\text{SAMPLE}}{\text{SECOND}}$. The signal has one of these three fundamental frequencies: $\{100, 150, 200\}$ Hertz. The period is one of the following three values:

$$\begin{aligned} P &= \left\{\frac{1}{100}, \frac{1}{150}, \frac{1}{200}\right\} \text{ sec} \\ N &= \left\{\frac{1200}{100}, \frac{1200}{150}, \frac{1200}{200}\right\} = \{12, 8, 6\} \end{aligned} \quad (5.7)$$

So to determine the fundamental frequency, we need only choose which of $\{r_y[6], r_y[8], r_y[12]\}$ is largest–comparison with $r_y[0]$ is unneeded.

To do this in Matlab, use

```
[Q I]=max(abs(R([7:13]));N=I+6
```

Note that if `abs(R(7))` is the largest, Matlab will give `I=1`, since `R(7)` is the first value of `R(7:13)`. So we must add to `I` the lower limit 6. And, as always, we must add 1 to everything, since Matlab indexing starts at 1, not 0.

To ensure that the peak is identified, the maximum should be taken over the complete range `[7:13]`, not just the three values `[abs(R(7)),abs(R(9)),abs(R(13))]`.

This same idea can be applied to fundamental frequency identification.

5.2.7 Choosing Among Signals

Suppose we have noisy observations $y[n]$ of one of three possible signals $\{x_1[n], x_2[n], x_3[n]\}$:

$$\begin{aligned} y[n] &= x_1[n] + w[n] \text{ OR} \\ &= x_2[n] + w[n] \text{ OR} \\ &= x_3[n] + w[n]. \end{aligned} \quad (5.8)$$

To determine which of the three signals is present, compute the three *correlations*

$$\begin{aligned} r_1 &= \sum y[n]x_1[n] \\ r_2 &= \sum y[n]x_2[n] \\ r_3 &= \sum y[n]x_3[n] \end{aligned} \tag{5.9}$$

and choose the largest of $\{|r_1|, |r_2|, |r_3|\}$. This idea extends to any number of signals.

5.3 Other Concepts

5.3.1 Additive Synthesis

A synthetic instrument playing a note at frequency f can be created mathematically using

$$x(t) = \sum_{k=1}^{K} c_k \cos(2\pi k f t) \tag{5.10}$$

for some choice of harmonics amplitudes $\{c_k\}$ that determine the timbre of the instrument. The synthetic trumpet was created in this way.

Your team will create its own synthetic instrument using additive synthesis by choosing some $\{c_k\}$. One way to do this is to find a sound you like, compute its line spectrum using the FFT, and read off the amplitudes of the largest lines to get $\{c_k\}$. Or you can just try different values until you come up with a sound you like.

The (subjective) criterion here is that your team has to agree that the sound is "cool."

Matlab code for generating your sound:

```
%D=duration of note in seconds.
%F=frequency of note in Hertz.
%C=row vector of 9 amplitudes.
T=[0:1/44100:D];%sampling rate=44100.
X=C*cos(2*pi*F*[1:9]'*T);
```

5.3.2 Reverb(eration)

When you listen to the University of Michigan marching band, you don't hear just one trumpet (unless they are playing a solo). You hear the combined sum of many trumpets. Of course, this makes their sound louder. But there is another effect: **reverb** (short for *reverberation*).

Because the trumpet players do not occupy the same exact point in space, you hear some trumpets a fraction of a second later than others. If the trumpet section is spread across 10 yards, and you are sitting in an end zone, you hear some trumpets (30 feet)/$[1050\frac{\text{FEET}}{\text{SEC.}}] = \frac{1}{35}$ second later than some other trumpets in the section.

You can get a similar sound by adding slightly delayed copies of the trumpet signal. This is the "singing in the shower" effect, or reverb.

Matlab code for reverbing the trumpet:

```
load trumpet.mat
LX=length(X);S=44100;D=0.01;K=5;
N=round(S*D);for I=1:K;M=N*I;
X=X+[zeros(1,M) X(1:LX-M)];
end;soundsc(X,S)
```

Try different values of the delay D and number of trumpets K until you find one your team likes.

5.3.3 Signal-to-Noise Ratio (SNR)

To evaluate the performance of your transcriber, you see how much noise you can add to it before it starts to fail a significant fraction of the time. The obvious performance metric to use is error rate (percent of the time your transcriber gives the wrong answer) vs. noise level.

But noise level by itself isn't enough; what counts is noise level vs. signal level. If your signal is a sinusoid with amplitude 100, noise with maximum value of one is a small amount of noise.

But if your signal is a sinusoid with amplitude 0.01, then that same noise is a large amount.

So it is the **ratio** of noise strength to signal strength that counts. Specifically,

- The strength of signal $x[n]$ is its *power* $\frac{1}{N}\sum_{n=1}^{N} x[n]^2$ where N is the length of $x[n]$. If $x[n]$ is periodic, then N is the period.
- The signal-to-noise ratio for a signal $x[n]$ and noise $v[n]$ is thus $\frac{\frac{1}{N}\sum_{n=1}^{N} x[n]^2}{\frac{1}{N}\sum_{n=1}^{N} v[n]^2}$.
- Signal-to-noise ratio is usually expressed in **decibels**. A bel is the logarithm (base 10) of something, so a decibel is $10\log_{10}$ of it. The **Signal-to-noise ratio in decibels** is
- $\mathrm{SNR}=10\log_{10}\frac{\frac{1}{N}\sum_{n=1}^{N} x[n]^2}{\frac{1}{N}\sum_{n=1}^{N} v[n]^2}$.

To evaluate the performance of your transcriber, generate a single signal (e.g., the trumpet playing note B) and do the following:

1. Add some noise with fixed strength;
2. Run the transcriber; check its output;
3. If wrong, increase #errors by one;
4. Do this 100 times for different noises;
5. The error rate=#errors/100 at that SNR;
6. Do 10 times for different noise strengths.

The skeleton of a program that can be used to evaluate the performance of your transcriber is:

```
[Generate note in X with synthesizer]
for S=1:10;NS=0;ER=0;
for I=1:100;%100 different noises.
N=S*5*randn(1,length(X));%noise level.
NS=NS+sum(N.*N)/100;%Noise strength.
Y=X+N;%Add noise to synthesizer note.
[Run transcriber on noisy note Y]
if [transcriber wrong];ER=ER+1;end
end;E(S)=ER;%errors at noise level S.
SNR(S)=10*log10(sum(X.*X)/NS);
end;plot(SNR,E)
```

See also the demo files `project3.m`.

Chapter 6

Performance of a System in Noise

6.1 Signal Processing System

We consider a simple touch-tone phone system which sends as its message a seven-digit phone number, e.g., "8675309." Each of the seven digits of the phone number is one of the nine digits $\{0, 1, 2 \ldots 9\}$. There is no hyphen present.

The i^{th} of the seven digits is represented by an integer x_i. If the phone number is "8675309," then $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ is x_1=8;x_2=6; etc.

6.1.1 Synthesizer

The goal of the synthesizer is to transform a phone number $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ into a continuous-time signal $x(t)$ that can be transmitted along a wire or through space.

The actual signal sent by the system is a 7-second-long analog (continuous-time) signal $x(t)$. $x(t)$ is defined as

$$\begin{aligned}
x(t) &= x_1 \quad \text{for} \quad 0 \le t \le 1 \\
x(t) &= x_2 \quad \text{for} \quad 1 \le t \le 2 \\
x(t) &= x_3 \quad \text{for} \quad 2 \le t \le 3 \\
x(t) &= x_4 \quad \text{for} \quad 3 \le t \le 4 \\
x(t) &= x_5 \quad \text{for} \quad 4 \le t \le 5 \\
x(t) &= x_6 \quad \text{for} \quad 5 \le t \le 6 \\
x(t) &= x_7 \quad \text{for} \quad 6 \le t \le 7
\end{aligned}$$

If the phone number is "8675309," then $x(t)$ is

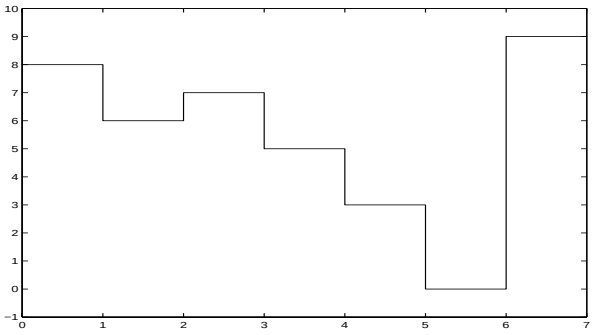


Figure 6.1: Signal for "8675309."

This figure was generated using Matlab code (yes, the extra 9 is required to make this plot)

```
clear;X=[8 6 7 5 3 0 9 9];
stairs([0:7],X),axis([0 7 -1 10])
```

It should be noted that actual touch-tone phone signals consist of two sinusoids. The frequencies of the pair of sinusoids specify which digit is sent. We consider a simpler scheme here to make things easier. We define

- $x(t)$=signal transmitted by synthesizer.
- $y(t)$=signal received by transcriber.

$y(t)$=$x(t)$. Later $y(t)$ will be $x(t)$ plus noise.

6.1.2 Transcriber

The goal of the transcriber is to convert the signal $y(t)$ received into a phone number $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$. The transcriber is:

1. *Sample* the signal so it can be processed using a computer. We use a sampling rate of $1000\frac{\text{SAMPLE}}{\text{SECOND}}$. The result is $y[n]$=$y(\frac{n}{1000})$.
2. *Segment* (i.e., divide up or partition) the received signal into seven segments of length 1 second=1000 samples each.
3. Transcribe each of the seven segments into a single digit by averaging the 1000 numbers in each segment. This gives seven averages.

The *estimated* first digit $\hat{x}_1$ is determined from the average $\bar{y}_1$ of the first 1000 numbers of $y[n]$:

$$\begin{aligned} \bar{y}_1 &= \frac{1}{1000}\sum_{n=0}^{999} y[n] \\ \hat{x}_1 = 0 &\quad \text{if} \quad \bar{y}_1 = 0 \\ \hat{x}_1 = 1 &\quad \text{if} \quad \bar{y}_1 = 1 \\ \hat{x}_1 = 2 &\quad \text{if} \quad \bar{y}_1 = 2 \\ \hat{x}_1 = 3 &\quad \text{if} \quad \bar{y}_1 = 3 \\ \vdots &\quad \text{if} \quad \vdots \end{aligned} \tag{6.1}$$

The *estimated* second digit $\hat{x}_2$ is determined from the average $\bar{y}_2$ of the *next* 1000 numbers of $y[n]$:

$$\begin{aligned} \bar{y}_2 &= \frac{1}{1000}\sum_{n=1000}^{1999} y[n] \\ \hat{x}_2 = 0 &\quad \text{if} \quad \bar{y}_2 = 0 \\ \hat{x}_2 = 1 &\quad \text{if} \quad \bar{y}_2 = 1 \\ \hat{x}_2 = 2 &\quad \text{if} \quad \bar{y}_2 = 2 \\ \hat{x}_2 = 3 &\quad \text{if} \quad \bar{y}_2 = 3 \\ \vdots &\quad \text{if} \quad \vdots \end{aligned} \tag{6.2}$$

The remaining five digits are estimated similarly.

It should be evident that this transcriber will work perfectly if $y(t)$=$x(t)$.

6.2 Performance in Noise

Once noise is added to $y(t)$, the transcriber will no longer work perfectly. The goal of noise analysis is to determine how noise added to the data degrades the system performance.

6.2.1 Additive Noise

There is always noise in any real-world system. The received signal $y(t)$ is the transmitted signal $x(t)$ plus added noise $v(t)$:

$$\begin{aligned} y(t) &= x(t) + v(t) \quad \text{before} \quad \text{sampling} \\ y[n] &= x[n] + v[n] \quad \text{after} \quad \text{sampling} \end{aligned} \tag{6.3}$$

A common model for noise is zero-mean *white Gaussian noise* (WGN). There are good reasons for this that you will learn in a statistics course.

Zero-mean white Gaussian noise can be generated in Matlab using `randn`. The signal plus noise for "8675309" is shown in Figure 6.2:

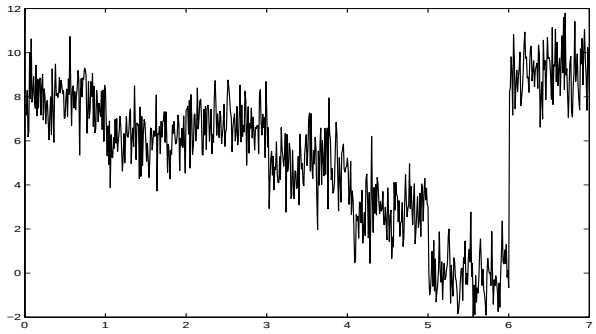


Figure 6.2: Noisy signal for "8675309."

This figure was generated using

```
clear;M=[8 6 7 5 3 0 9];
XX=ones(100,1)*M;X=XX(:);
Y=X+randn(700,1);
plot([1:700]/100,Y)
```

6.2.2 Transcriber for Noisy Data

The transcriber given in (6.1) and (6.2) is now modified. Instead of using the mean $\bar{y}_i$ of the 1000 numbers in the i^{th} segment to estimate x_i,

- Mean $\bar{y}_i$ is *rounded to the nearest integer*.
- If $\bar{y}_3$=7.3, then $\hat{x}_3$=7, since 7.3 rounds to 7.

The following code implements the synthesizer on the phone number specified in `M`, samples at $1000\frac{\text{SAMPLE}}{\text{SECOND}}$, and adds noise scaled by factor `S`:

```
clear;M=[8 6 7 5 3 0 9];
XX=ones(1000,1)*M;X=XX(:);
S=1;Y=X+S*randn(7000,1);
```

The following code implements the transcriber on the noisy data stored in column vector `Y`:

```
YY=reshape(Y,1000,7);
XHAT=round(mean(YY))
```

6.2.3 Performance Measure

How do we measure how well the transcriber is working on noisy data? We need a *performance measure*, which is a number that tells us how well the transcriber is working (business people call this a "metric").

An obvious performance measure is the *single-digit error rate*, which is the percent of the time that the transcriber gets a single digit wrong at a given noise level. For example, if the transcriber is wrong 12% of the time at a given noise level, the error rate is 12% at that noise level.

Another performance measure is the *phone number error rate*, which is the percent of the time that the transcriber gets any of the seven digits in a phone number wrong. Since this (and other measures) can be calculated from the single-digit error rate, we use the single-digit error rate as our performance measure.

How do we know that the transcriber is wrong 12% of the time? A simple way to *estimate* the error rate is to proceed as follows:

1. Generate the noiseless sampled synthesizer signal $x[n]$ for a single digit. Note that $x[n]$ is one second=1000 samples long.
2. Generate 100 different random noise signals, each having the same level. Matlab can do this using a random number generator.
3. Form 100 different noisy signals by adding the 100 noise signals (one at a time) to the noiseless synthesizer signal.
4. Run the transcriber on each of the 100 noisy signals, getting 100 digit estimates.
5. Count the nuber of times e that the transcriber output the wrong digit. The error rate at that noise level is then $\frac{e}{100}$=e%.

The more noise signals we use, the more accurate this estimate of the error rate will be. "100" is a commonly-used figure, for reasons that you will learn in a statistics course.

6.2.4 Signal-to-Noise Ratio (SNR)

We then run the above procedure several times, using different noise levels. But note that what really matters is not the actual noise level, but the noise level *relative to the signal level*. A noise level of one is high if the signal level is 0.01, but low if the signal level is 100.

We also must define noise and signal levels. This is usually done using the *power* (average of squared values) of a signal. The power of a sampled signal $x[n]$ having a length of 1000 is

$$\text{Power of } x[n] = \frac{1}{1000}\sum_{n=0}^{999} x[n]^2. \qquad (6.4)$$

The *signal-to-noise ratio* (SNR) is then the ratio of signal power to noise power, expressed in decibels (dB) by taking the common (base=10) log ("bels") and multiplying by 10 ("decibels"):

$$\text{SNR} = 10\log_{10}\frac{\frac{1}{1000}\sum_{n=0}^{999} x[n]^2}{\frac{1}{1000}\sum_{n=0}^{999} v[n]^2}. \qquad (6.5)$$

The factors $\frac{1}{1000}$ cancel, so they can be omitted.

6.2.5 System Performance

The following Matlab program computes and plots single-digit error rate vs. SNR for ten different SNR's, using 100 trials to generate each of ten points on the graph.

```
%Synthesizer for digit M:
clear;M=7;X=ones(1,1000)*M;
%Noise performance program:
for S=1:10;NS=0;ER=0;%10 noise levels
for I=1:100;%100 noise signals
N=S*5*randn(1,length(X));%scale noise
NS=NS+sum(N.*N)/100;%avg.  noise power
Y=X+N;%add noise to signal
%Transcriber:
XHAT=round(mean(Y));
%Noise performance program, cont.:
if(XHAT==M);ER=ER+0;else ER=ER+1;end
end;E(S)=ER;%error rate@noise level
SNR(S)=10*log10(sum(X.*X)/NS);
end;plot(SNR,E)
```

This same program can be used for your music transcriber. Replace the digit transcriber with your music transcriber, with these changes:

- Use your music synthesizer to generate a single musical note (instead of a digit).
- Use your music transcriber to generate an estimated musical note (instead of a digit).
- You *don't* need the MIDI-to-staff musical notation step of your transcriber. Just use the MIDI representation of your note.
- Make sure that your transcriber now accepts as input `Y` instead of `X` (check your code).
- Make sure your program doesn't use `N` twice, to represent both #signals and the noise.

Figure 6.3 is a typical result. Note that running the above program several times will produce plots similar, but not identical, to this.

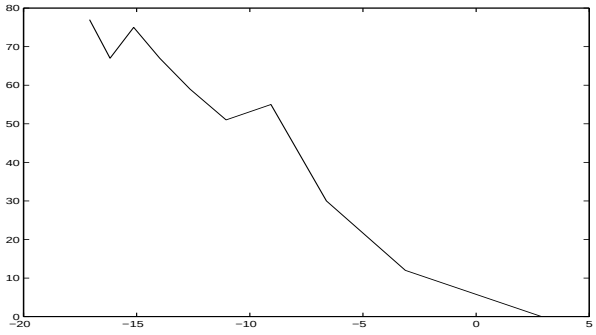


Figure 6.3: Transcriber Performance

Any customer of your engineering product (phone or music transcriber) will want to see this plot. The customer must provide specs for noise level and acceptable error rate. For example, if a 2% error rate is acceptable, how high a noise level can your transcriber handle?