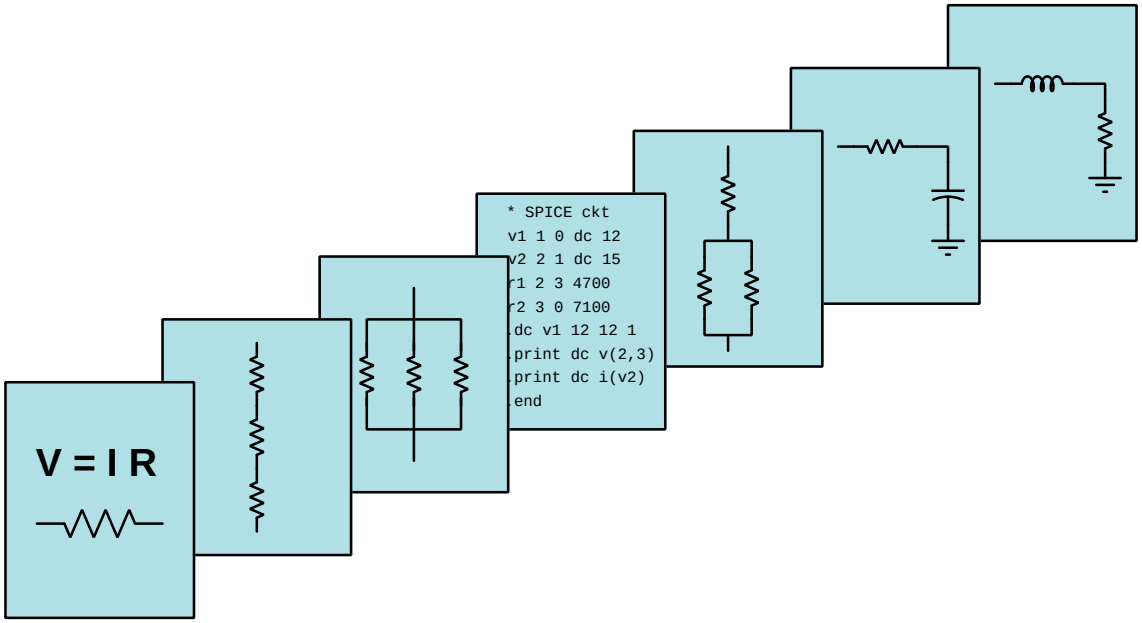
Modular Electronics Learning (ModEL) project



Frequency-Domain Analysis

Last update = 2 October 2024

Contents

Chapter 1

Introduction

1.1 Recommendations for students

The consideration and analysis of electrical signals from the perspective of the *frequency domain* is one of the more important principles in the study of AC circuits, particularly electronic communication systems. Instead of considering signals as voltage or current values varying in amplitude over a span of *time*, we now also consider these same signals as voltage or current values varying in amplitude over a span of *frequency*. One of the interesting consequences of considering signals from the perspective of frequency rather than time is that it allows us to equate signals of arbitrary wave-shape in terms of perfect sinusoids (sine-wave-shaped signals).

Important concepts related to the frequency domain include **spectra**, light **color** as it relates to **frequency**, **oscilloscopes** and **oscillographs**, mathematical **sine** and **cosine** functions, **Fourier's Theorem**, **fundamental** frequency, **harmonic** frequency, **superposition** of waveforms, **linearity**, **noise** and **noise floor**, and signal **distortion**.

Here are some good questions to ask of yourself while studying this subject:

- How might an experiment be designed and conducted to prove the existence of harmonic frequencies within a non-sinusoidal waveform, without the benefit of a spectrum analyzer? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- How might an experiment be designed and conducted to prove the existence or non-existence of certain harmonic frequencies within a specific non-sinusoidal waveform? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What does a transparent prism do to light?
- What is "monochromatic" light, and how does it differ from "white" light?
- What is meant by the term "decomposition" as it applies to light or to electrical signals?

- How do we define "frequency" for any signal?
- What does a spectrum analyzer do that an oscilloscope does not do?
- How do sinusoidal signals of different frequency appear on an oscilloscope?
- How do sinusoidal signals of different frequency appear on an spectrum analyzer?
- What is a *harmonic* frequency, and how does it relate to the *fundamental* frequency?
- What does it mean to say that a waveform is "equivalent" to a harmonic series?
- What are some practical examples of *periodic* signals?
- What are some practical examples of *non-periodic* signals?
- What practical applications exist for frequency-domain analysis of signals?
- Which harmonics are present, and how strong for each, in a pure sine-wave signal?
- Which harmonics are present, and how strong for each, in a square-wave signal?
- Which harmonics are present, and how strong for each, in a triangle-wave signal?
- Which harmonics are present, and how strong for each, in a sawtooth-wave signal?
- What factor determines the harmonic content of a pulse signal?
- What type of signal contains *all* frequencies?
- What type of signal contains *no* harmonics at all except for the fundamental?
- How does the steepness of a waveform's rise and fall times relate to its harmonic content?
- How does the symmetry of waveform above versus below its centerline relate to its harmonic content?
- How may a waveform's harmonic content may be analyzed using filter networks?
- How may a waveform's harmonic content may be analyzed using a mixer and a voltage-controlled oscillator?
- How may a waveform's harmonic content may be analyzed using a simple computer algorithm?

1.2 Challenging concepts related to frequency-domain representation of signals

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Fourier's Theorem** – while this is no doubt a non-intuitive concept, it is incredibly useful. Knowing that any waveshape whatsoever may be reproduced by summing together the right combination of sinusoidal waves may defy our intuition, but at least seeing it is possible to synthesize common non-sinusoidal waveshapes like square and triangle using nothing but sine and/or cosine waves helps prove one can get non-round waves from lots of round waves added together. The practical upshot of this is that is it possible to consider very complex waveshapes as being nothing more than a set of sine waves added together. Since sine waves are easy to analyze in the context of electric circuits, this means we have a way of simplifying what would otherwise be a dauntingly complex problem: analyzing how circuits respond to non-sinusoidal waveforms. Time spent with a simple oscilloscope and spectrum analyzer viewing signal generator waveforms is also helpful in grasping how time-domain and frequency-domain representations relate.

The *Programming References* chapter contains multiple examples of simple programs written in C++ demonstrating various ways of decomposing arbitrary waveforms into their Fourier series. The *Historical References* chapter also contains examples of legacy technology doing the same, from simple inductor-capacitor band-pass filter networks tuned to known harmonic frequencies, or metal reeds trimmed to resonate at unique frequencies, these old technologies show us the same fundamental truth of waveforms that modern spectrum analyzers do.

1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

- **Outcome** – Apply heuristic rules to associate waveform versus spectrum characteristics

  Assessment – Predict harmonic content from time-domain views of AC signals; e.g. pose problems in the form of the "Discerning even/odd harmonics from the time domain" Diagnostic Reasoning question.

- **Outcome** – Independent research

  Assessment – Locate amplifier datasheets and properly interpret some of the information contained in those documents related to frequency-domain representation of signals including Total Harmonic Distortion (THD), spectrum measurements of sinusoidal test signals, etc.

Chapter 2

Case Tutorial

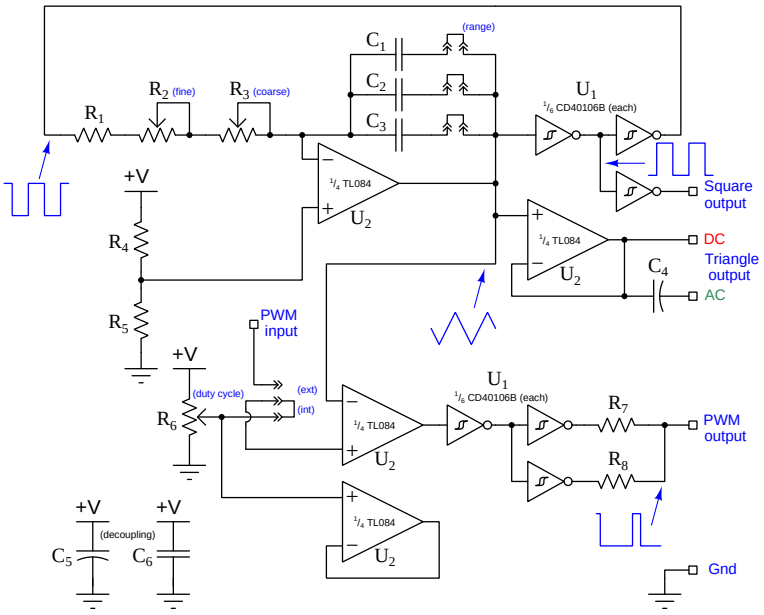
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

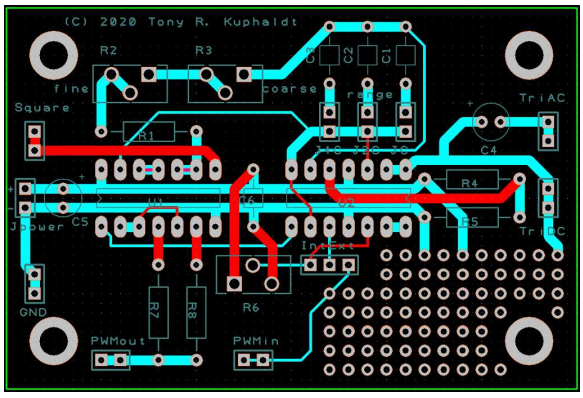
These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: sine versus non-sine AC sources

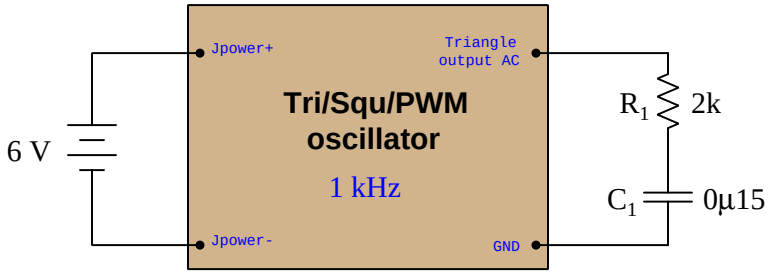
Students learning to analyze RLC networks powered by AC voltage sources typically rely on expensive signal generators to produce the pure sine-wave AC excitation voltage necessary for voltmeter measurements to closely match predictions. However, robust triangle-wave oscillator circuits are much less complicated to design and build than sine-wave oscillator circuits, so if students wish to build their own signal generators for these introductory AC experiments it is good to know that triangle-wave excitation yields results very close to sine-wave excitation.

A simple and versatile signal generator circuit appears below, outputting triangle, square, and PWM (pulse-width-modulated) signals. The first image is the schematic diagram, followed by a PCB layout:





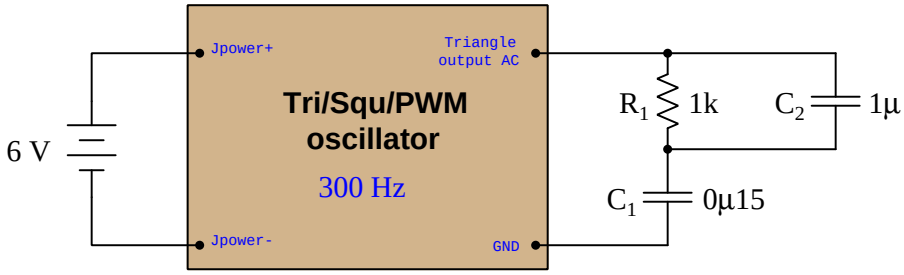
Here are some test results on simple RC networks:



Connected to the RC network, frequency was measured using a Fluke model 87-III multimeter and adjusted to 1 kHz, and then total voltage measured across the series $R_1 \leftrightarrow C_1$ combination as 233.0 mVAC.

Parameter	**Measured** (triangle-wave)	**Predicted** (sine-wave)
V_{R1}	205.7 mVAC	205.8 mVAC
V_{C1}	109.3 mVAC	109.2 mVAC

Testing a slightly more complex circuit at a frequency of 300 Hz, the loaded voltage output of the oscillator being 231.6 mVAC this time:



Parameter	**Measured** (triangle-wave)	**Predicted** (sine-wave)
V_{R1}	23.7 mVAC	27.43 mVAC
V_{C1}	207.5 mVAC	207.0 mVAC
V_{C2}	23.7 mVAC	27.43 mVAC

In both applications, the greatest error between measured voltage and predicted voltage as a percentage of total voltage was in the second circuit across $R_1||C_1$ (23.7 milliVolts rather than 27.43 milliVolts), and this is only -1.61% of the source voltage which is considerably less than the $\pm$ 5% tolerance of the resistor and capacitors!

If we compare the Fourier series for a sine wave and a triangle wave (both having unity peak values and a frequency of ω) we see that the first harmonic of the triangle wave function is identical to the sine wave, and that all the other harmonics in the triangle wave are significantly smaller-amplitude than the fundamental:

Sine wave

$$\cos \omega t$$

Triangle wave

$$\cos \omega t + \frac{1}{9} \cos 3\omega t + \frac{1}{25} \cos 5\omega t + \frac{1}{49} \cos 7\omega t + \cdots + \frac{1}{n^2} \cos n\omega t$$

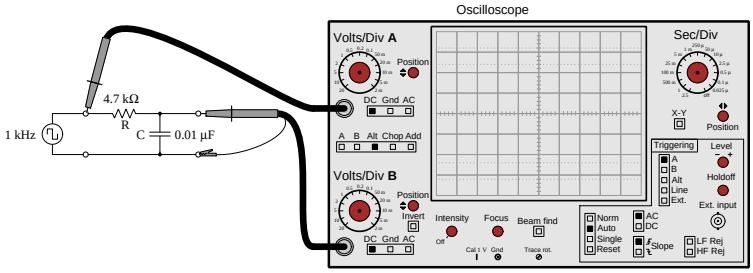
This tells us any deviations between the measured (triangle-wave) and predicted (sine-wave) voltage values are likely to be minimal, the third harmonic being only 11.1% of the fundamental's amplitude, the fifth harmonic being only 4% of the fundamental's amplitude, etc. The effects of higher-order harmonics are truly negligible due to their vastly smaller amplitudes as well as due to the fact that most digital multimeters suffer "cut off" in the audio-frequency range and therefore cannot measure signal components in the tens of thousands of Hertz.

With access to a digital oscilloscope having FFT capability[1] to show precise voltage values for each harmonic of a measured waveform, we have an even better solution for obtaining voltage measurements in agreement with predicted values when not using perfectly sinusoidal signal generators. Since the oscilloscope's FFT algorithm separates and displays each of the sinusoidal harmonics apart from one another in any non-sinusoidal waveform, if we simply pay attention to the magnitudes of a common harmonic frequency within each voltage measurement we will essentially take circuit measurements on purely sinusoidal voltages of the same frequency. For example, we could measure the fundamental (i.e. the first harmonic)[2] amplitude of source voltage, then the fundamental amplitudes of each of the other components' voltages, and check to see that these measured voltage values match well with our predictions at that frequency. This technique, in effect, lets us measure the effects of a purely sinusoidal signal even when the real signal is not sinusoidal at all, by taking measurements only on a common harmonic of the measured voltages!

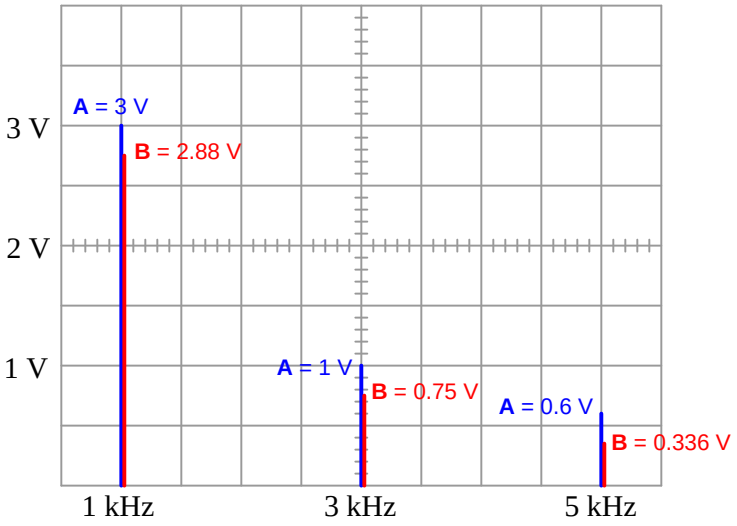
[1] At the time of this writing (2022) some inexpensive oscilloscopes may be found with rather poor FFT resolution, resulting in wide spectral peaks with uncertain height (voltage) values. You know you are working with a sufficiently precise instrument when the harmonic peaks show as thin lines rather than exaggerated bell-curves.

[2] There is no particular reason why we might choose the first harmonic over any of the others, other than the fact that with triangle and square waves this fundamental will be vastly stronger than any of the other harmonics.

This testing technique deserves some elaboration, and so we shall explore it by example. Consider the following test circuit where a signal generator configured to output a square-wave AC signal at 1000 Hz energizes a simple RC network consisting of a 4.7 kΩ resistor and a 0.01 μF capacitor:

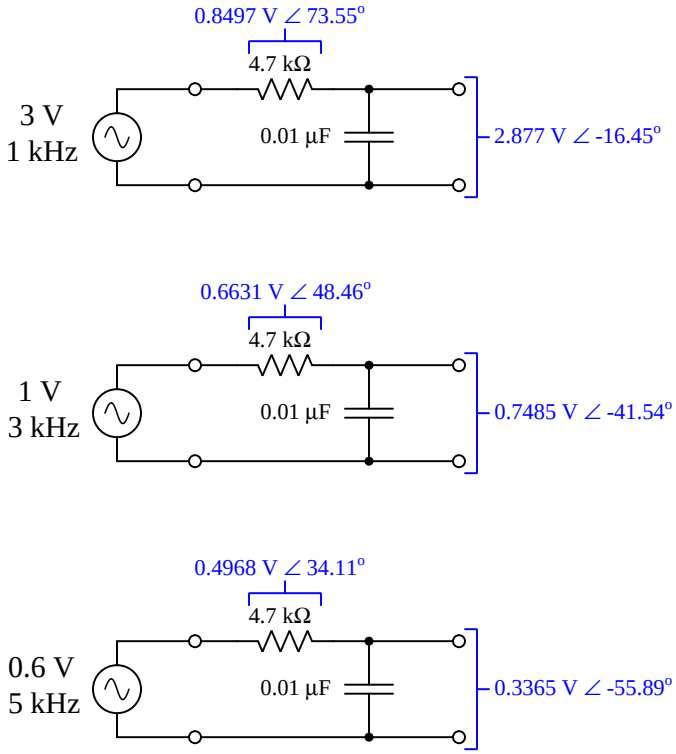


We know from Fourier analysis that a square wave is actually equivalent to a sine wave at the same fundamental frequency added to another sine wave one-third the amplitude at three times that frequency (3rd harmonic) added to another sine wave one-fifth the amplitude of the fundamental at five times that frequency (5th harmonic), and so on. If we examine the frequency-domain plots of the signal generator's output (channel A) versus the capacitor's voltage drop (channel B), we see the circuit's response to pure sine waves at each of those frequencies:



The relative peak heights of the channel A signal (3 Volts, 1 Volt, 0.6 Volts) are simply the result of the Fourier series for a square wave and has nothing to do with the RC network. The ratios between peak heights of channel A and channel B at each harmonic frequency, however, are unique to the 4.7 kΩ and 0.01 μF RC network because those voltage pairs represent the attenuation of this particular network at each of those sinusoidal frequencies.

If we mathematically analyze this same RC network for each of the square wave's harmonic amplitudes and frequencies used in the test circuit, we should obtain results verifiable by using the oscilloscope in FFT mode:

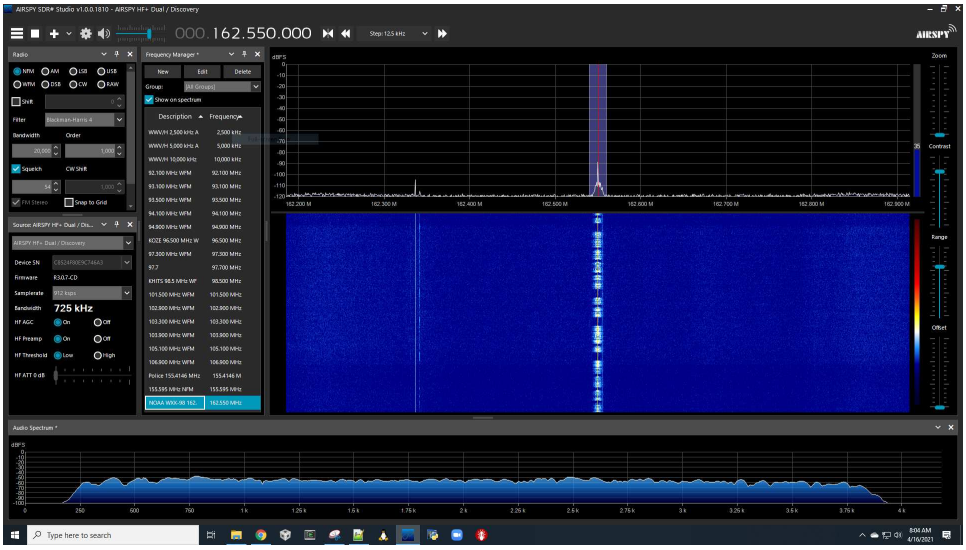


For students with access to oscilloscopes having fine-resolution FFT capability, this not only means it is unnecessary to secure a signal generator with pure sinusoidal output, but it also means the ability to energize any AC circuit with *any* waveshape and test its response at multiple sinusoidal frequencies *simultaneously*!

2.2 Example: SDR spectrum displays

A modern technology for monitoring radio transmissions is something called *Software-Defined Radio* or *SDR*. This is a radio receiver circuit that connects to a digital computer, sending that computer a fairly raw stream of data representing signals received by the antenna, and leaving it up to software algorithms in that computer to de-modulate and otherwise de-code those signals to reveal intelligible information. One of the prominent features of SDR is its ability to display the spectrum of received radio signals, both as a frequency-domain plot and something called a *waterfall* display.

A screenshot of SDR software showing both the frequency-domain plot and waterfall display is appears below. The frequency-domain plot has a vertical axis calibrated in dBFS (decibels of full-scale) and a horizontal axis calibrated in Hertz. The waterfall plot is immediately below the frequency-domain plot, and is currently a background of blue with some lighter-colored vertical stripes. It shares the same horizontal scale as the frequency-domain plot (Hertz) but its vertical axis is a slow-moving time scale and signal intensity is represented by *color*:



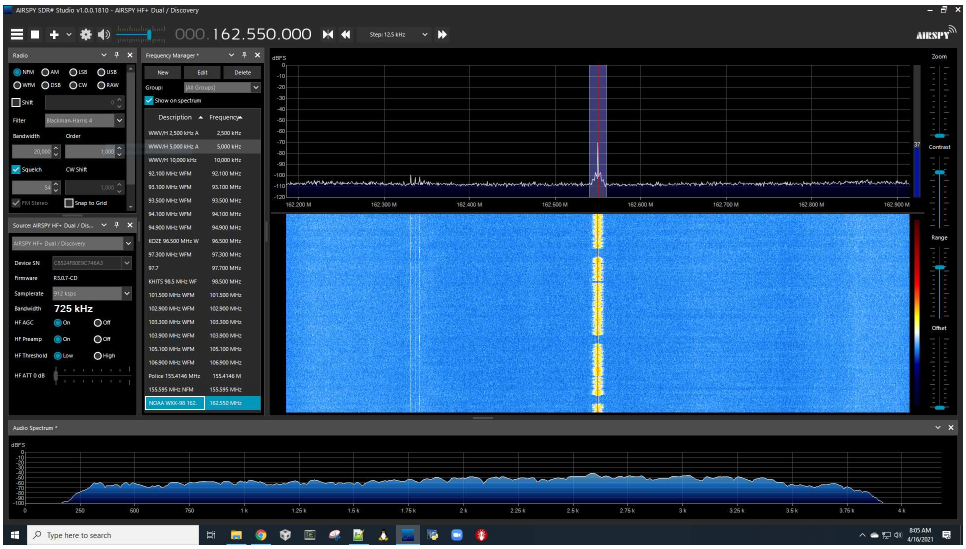
In this screenshot we see the SDR software centered on a signal peak representing a local NOAA[3] weather broadcast transmitter at a frequency of 162.55 MHz. Like all frequency spectrum plots, signal amplitude is shown by the height of the plot, in this case the carrier frequency signal measuring just above −90 dBFS. The red vertical line and blue-grey color band overlaid on the frequency-domain plot show the SDR software's center (tuning) frequency and the bandwidth of its digital filter for selecting this particular station. The waterfall display shows a colorized representation of signal amplitude, that colorize plot slowly "falling" down over a period of about 20 seconds from the top of the waterfall display to the bottom. This is why the middle stripe of color appears to be broken: the "breaks" represent periods where there is no audio signal and the NOAA transmitter

[3] In the United States, this is the National Oceanic and Atmospheric Administration, a federal agency tasked with ongoing study and reporting of terrestrial and aquatic conditions on Earth. Among its many functions is to provide regular weather reports and forecasts as part of its National Weather Service, this particular radio transmitter being one of those services.

outputs only a single carrier frequency at 162.55 MHz; the wider colored sections represent periods where speech occurred and the transmitter's signal occupied a wider slice of the frequency domain.

For this screenshot an extremely short antenna was connected to the input connector of the SDR unit, and as a result all signals were fairly weak. This included the *noise floor*, shown in the frequency-domain plot at about −120 dBFS and represented in the waterfall display as a solid blue background.

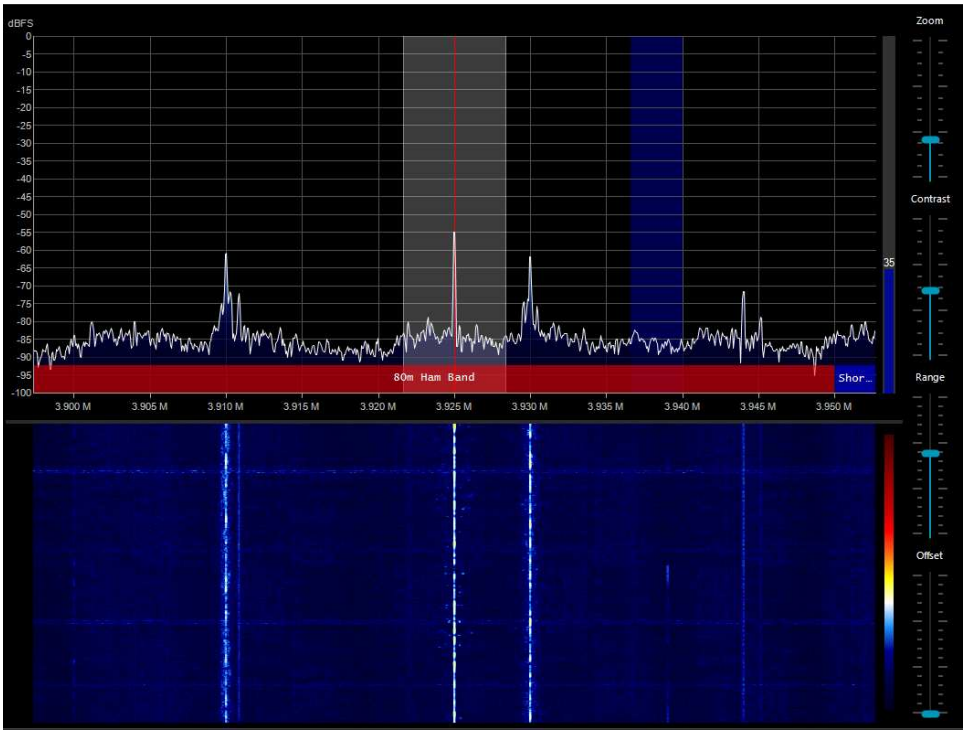
In this next screenshot I show the same SDR radio unit with SDR software locked into the same NOAA radio transmitter at 162.55 MHz. However, this time I touched the short antenna wire with my fingers, using my body as an antenna to capture more signal for the receiver to process. As a result we see all signals on the spectrum become stronger, with the noise floor moving up to approximately −110 dBFS:



Correspondingly, the waterfall display is uniformly brighter than before, that brightness representing increased signal strength across the displayed spectrum. Not only is the background a lighter shade of blue (with speckles representing peaks of random noise from the noise floor), but the NOAA station's signal is trending toward yellow and red.

In both examples we also see an audio spectrum display at the bottom of the SDR software display, representing the audio signal's frequency spectrum resulting from demodulation of the radio station's RF (radio-frequency) signal. As you can see, this audio spectrum extends from 0 Hz to 4 kHz, which is wide enough for typical human speech.

In this next screenshot we see an interesting phenomenon recorded by the SDR's waterfall display, a series of lighter-blue horizontal lines every so often:



Recall that the coloring of a waterfall display represents signal strength, with dark representing weak signal (or noise floor) and light representing strong signal. This is why the four peaks on the spectrum display correspond to four matching light-colored vertical lines on the waterfall display: the slowly-scrolling waterfall display results in each of these peaks tracing its own light-colored line on the waterfall. What *doesn't* seem to match the spectrum display, however, are the multiple *horizontal* lighter-blue lines we see on the waterfall display. These represent brief moments in time where the entire "noise floor" elevated to a higher signal-strength level. What could cause that to happen?

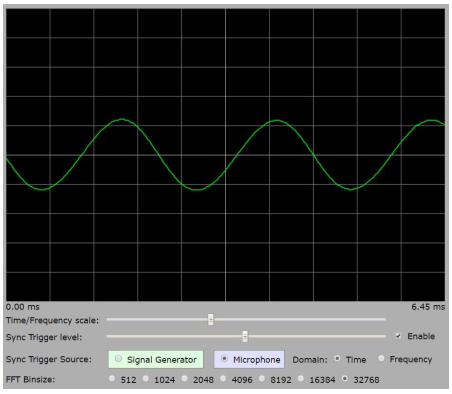
The answer to this question is a series of *lightning strikes* happening at that time. Bolts of lightning, of course, are short-duration impulses of high electrical energy. Since each lightning bolt's time duration is extremely brief, each lightning bolt is a nearly-perfect *delta impulse function* which is mathematically equivalent to a broad range of simultaneous frequencies. The SDR receiver displays each of these broadband noise bursts for what it is, resulting in a record of horizontal lines in the waterfall display.

Chapter 3

Simplified Tutorial

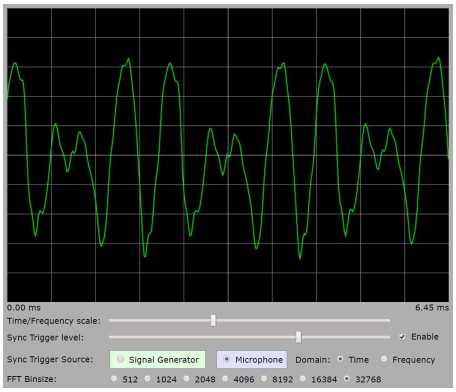
A great deal of electricity and electronics centers around *waves*: oscillations of some quantity, usually a voltage or a current, over time. Waves are at the very heart of *alternating current* (AC) circuits where both voltage and current rise and fall over time, never settling to a stable value. Many forms of information such as human speech, music, and serial digital data similarly take the form of waves in electronic circuits, the oscillating voltage and/or current quantities representing that information. Moreover, a vast range of physical phenomena beyond the scope of electric or electronic circuits take the form of waves, including tidal fluctuations, weather patterns, machine vibrations, and even light.

One way to represent a wave is to plot its changing value over time in the form of a graph. An example is shown here, taken from a microphone recording of a *tuning fork* vibrating after being physically struck:



We often refer to this particular wave-shape as being a *sine wave* because it follows the same pattern as the trigonometric "sine" function. Cosine waves also have this same shape, the only difference between a cosine wave and a sine wave being a shift in phase of 90 degrees or $\frac{\pi}{2}$ radians. If all we mean to describe is this particular shape of wave, we may use the word *sinusoidal* which means "like a sine wave".

Many natural phenomena oscillate in purely sinusoidal patterns. The voltage and current output by an AC generator when its shaft rotates is naturally sinusoidal. However, most oscillations found in the world are not sinusoidal but instead follow more complex patterns. Here is an example of a non-sinusoidal sound wave as recorded from a reed instrument called a *melodica*:



As you can see, the sound waves produced by a melodica rise and fall over time, but they do not do so in the same smooth, simple pattern we saw with the tuning fork.

Sinusoidal waves are relatively simple to quantify mathematically, and when we analyze circuits energized by sinusoidal voltages and currents the mathematical calculations are fairly straightforward. However, if we must analyze a circuit energized by voltages and currents following a more complicated wave-shape, the formulae used for analyzing sinusoidal-energized circuits don't work.

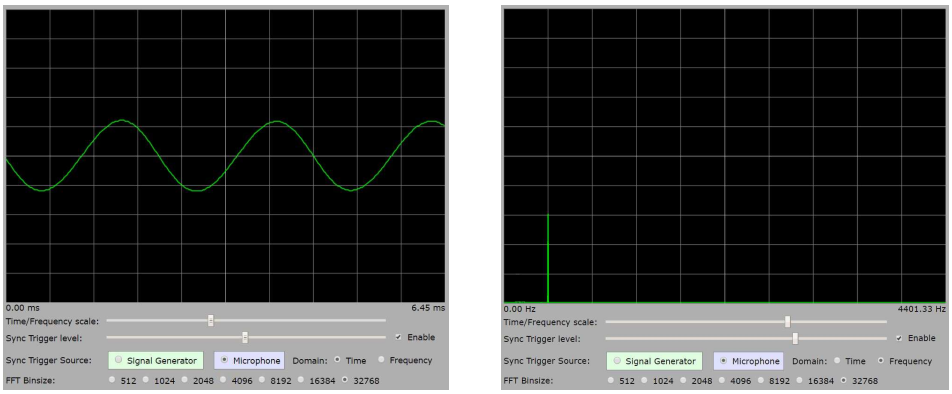
The French mathematician Jean Baptiste Joseph Fourier made a remarkable discovery (publicized in 1807) when he deduced that any wave-shape, no matter how complicated, is actually equivalent to a series of simple sinusoidal waveforms all added together. This is now known as *Fourier's Theorem*, and its important to us in analyzing circuits lies in the fact that any oscillating voltage or current is representable as a sum of sine and/or cosine waves at specific magnitudes and frequencies.

Let's pause for a moment to appreciate the significance of Fourier's Theorem. Not only does Fourier's Theorem tell us that we may synthesize any wave-shape we might want by adding together a collection of sinusoidal waves at just the right frequencies and magnitudes, but it also tells us any wave-shape we might encounter may be decomposed into a collection of sinusoidal waves. That is to say, the equivalence between a non-sinusoidal wave and a specific summation of sinusoidal waves works both ways – the two things are fully indistinguishable from each other. Fourier's Theorem in essence tells us that every wave-shape imaginable has a "recipe" where the only ingredients are plain sine and cosine waves.

To express this fact using the musical examples of the tuning fork and the melodica seen previously, this means it's theoretically possible to mimic the sound of a melodica by simultaneously striking the right combination of tuning forks, each fork tuned to a particular frequency and each fork struck with a particular amount of force following the appropriate "recipe"! I say that this is *theoretically* possible because to actually synthesize the sound of a melodica using tuning forks would require an entire *orchestra* of tuning forks with very unusual frequency specifications and ultra-precise striking forces – a difficult feat to achieve in practice.

Various mathematical techniques exist to determine precisely which sine/cosine wave magnitudes and frequencies comprise any sampled wave-shape. This topic is far too detailed and complex to discuss in this Simplified Tutorial, but suffice it to say that instruments exist for this purpose called *spectrum analyzers*[1]. The graphic display of a spectrum analyzer does not show the amplitude of the wave over time as an oscilloscope does, but rather shows each sinusoidal frequency as a peak in the graph, the horizontal location of each peak corresponding to the frequency of that sinusoid and the vertical height of each peak representing that sinusoidal wave's magnitude.

Below we see a pair of graphs representing the sound waves generated by a struck tuning fork. On the left is the *time-domain* graph displayed by an oscilloscope, and on the right is the *frequency-domain* graph displayed by a spectrum analyzer:

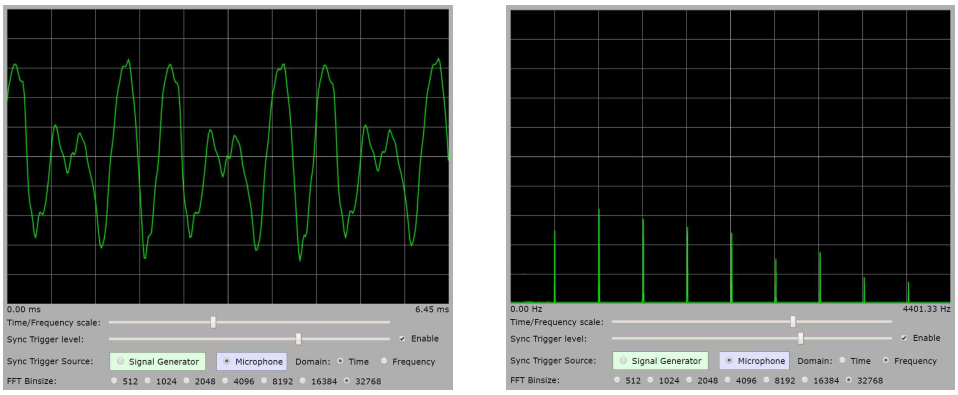


Note how the left-hand graph shows the oscillating sound wave plotted over time with the graph's horizontal axis (it's "domain") labeled in units of milliseconds, and note how the right-hand graph shows a single peak located at 440 Hz with the horizontal axis labeled in units of Hertz (Hz). Both of these graphs describe the sound produced by this tuning fork, but they describe it in different ways: one as a plot of sound pressure as a function of time, and the other as a plot of sound pressure as a function of frequency.

Only one peak exists in the tuning fork's frequency spectrum because its time-domain function is a pure and simple sinusoid. To use the "recipe" analogy again, the tuning fork's sonic recipe contains only one ingredient.

[1] Most modern digital oscilloscopes also offer spectrum-analysis capability, which is especially useful because this means one may plot both the time-domain and frequency-domain graphs on the same instrument!

If we examine the melodica's tone in both the time (left) and frequency (right) domains, we see the following results:



We've seen the time-domain (oscillograph) plot already as an example of a "complex" waveform contrasted against that of the tuning fork's, but the frequency-domain (spectrum analysis) is new to us. Note how the latter contains multiple peaks, each of them representing a different sinusoidal frequency comprising the complex sound of the melodica. Also, note how all of these peaks are equally spaced from each other which tells us their frequencies are whole-numbered multiples of the lowest frequency (in this case, 440 Hz, same as the tuning fork). Those peaks in the frequency domain (spectrum) plot tell us the "recipe" for the melodica's tone, consisting of multiple sinusoidal "ingredients" at specific frequencies and intensities.

As with the tuning fork's time-domain and frequency-domain plots, what we see above are just two different ways of representing the same sound made by the melodica.

Fourier's Theorem doesn't just tell us that any wave-shape is equivalent to a summation of sine and/or cosine waves, but also states that any *periodic* (i.e. repeating) wave-shape consists solely of sine/cosine waves having frequencies that are integer-multiple to each other, called *harmonic frequencies*[2]. The lowest of these harmonic frequencies is called the *fundamental frequency*, or also the *first harmonic frequency*. In this particular case of the melodica's tone, the fundamental frequency happens to be 440 Hz, so the harmonic frequencies must be:

- 1st harmonic (fundamental) = 440 Hz
- 2nd harmonic = 880 Hz
- 3rd harmonic = 1320 Hz
- 4th harmonic = 1760 Hz
- 5th harmonic = 2200 Hz
- etc . . .

[2] Alternatively, these harmonic frequencies may be called "fundamental" and "overtones". Confusingly, the first overtone is the second harmonic, the second overtone the third harmonic, etc. In acoustic and musical studies the overtones are sometimes called *partials*.

Many practical applications exist for representing oscillating quantities in the frequency domain rather than (or in conjunction with) the more familiar time domain:

- As alluded earlier, the mathematics of sinusoidal AC voltages and currents is fairly simple, so if we happen to know the Fourier "recipe" for a more complex AC voltage or current wave that tells us how many harmonics there are, what their frequencies are, and how large (Volts or Amperes) each one's magnitude is, we may analyze any circuit's response to that complex AC voltage or current simply by repeated analysis of each harmonic's effect on the circuit.

- A frequency-domain plot of a waveform offers a very clear and easy-to-understand way of detecting *distortion* in the wave. For example, if the wave-shape is supposed to be perfectly sinusoidal, then its frequency should only contain a single peak. However, if that supposedly perfect sinusoid is in fact distorted at all, we will plainly see this distortion as additional harmonics in the spectrum. Mild amounts of distortion are surprisingly difficult to visually detect in a time-domain oscillograph plot of a (mostly) sinusoidal wave.

- Frequency-domain plots also make it easier for us to spot certain informative features of a waveform that would be difficult or impossible to identify in the time domain. For example, being able to pinpoint *noise* present in a waveform, being able to discern the presence of intelligible information amidst competing signals, etc.

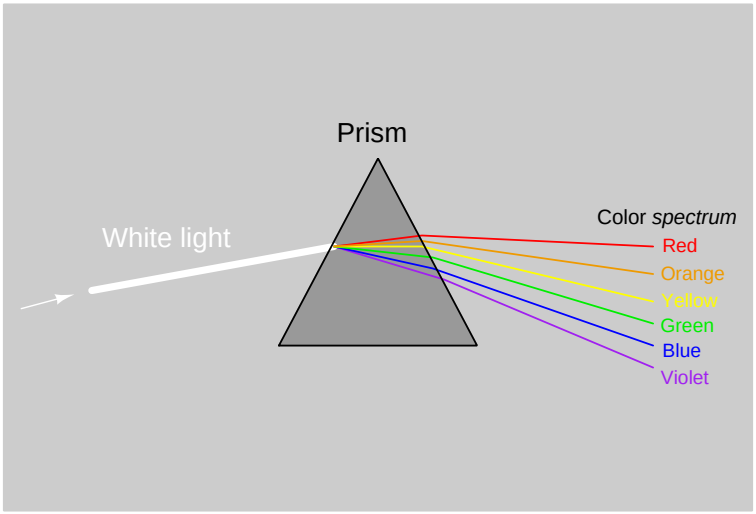
Returning to the "recipe" analogy again, frequency-domain instruments such as spectrum analyzers allow us to un-do a fully cooked or baked dish into its constituent ingredients to see what it's made of, and this can be a very powerful diagnostic tool.

Chapter 4

Full Tutorial

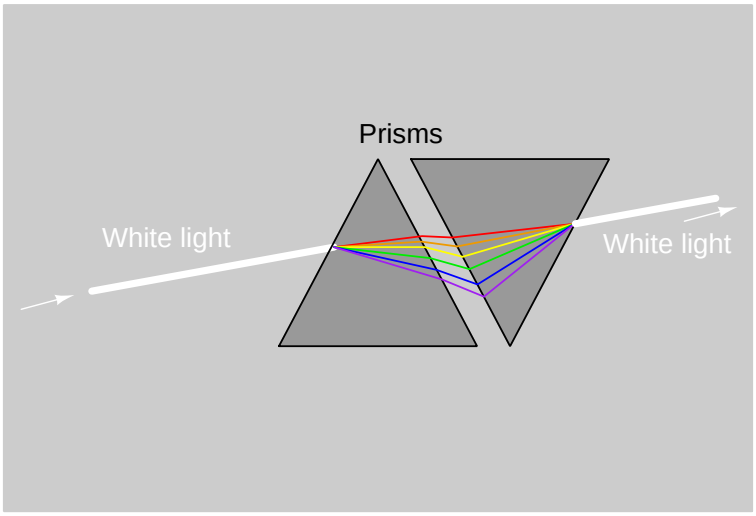
4.1 Composition of light

An optical *prism* is a transparent structure, usually made of clear plastic or glass, designed to bend light beams. When a beam of white light passes through a prism, it splits into beams of differing color called a *spectrum*:

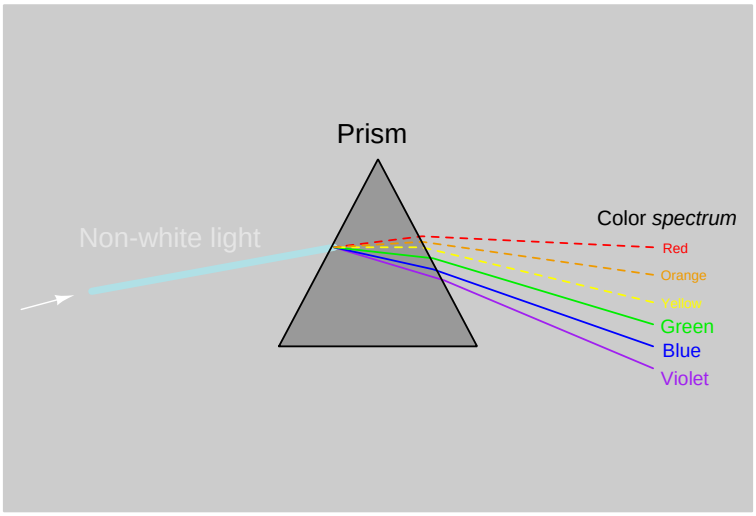


If the light beam introduced to the prism is purely white, the resulting spectrum will consist of colors all having equal intensity as shown in the above illustration. We could say that white light *consists* of all those colors mixed together in equal proportion.

Furthermore, we may demonstrate how white light may be reproduced from a full spectrum by using a second prism, demonstrating that the process of separating white light into its constituent colors is reversible:

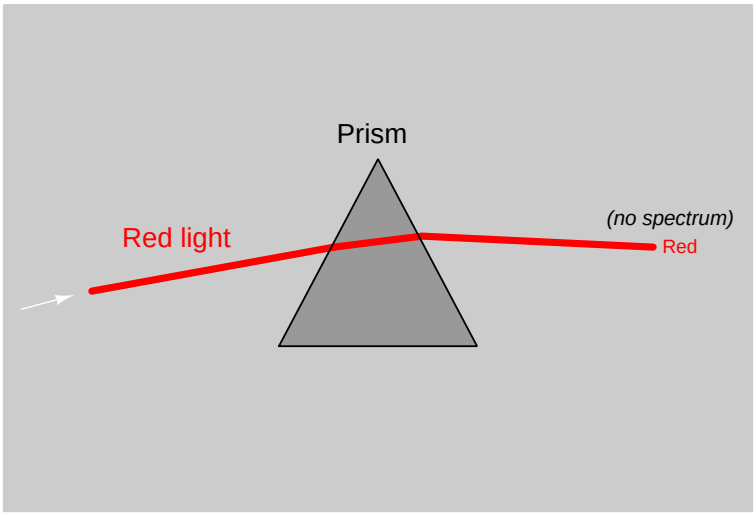


If the incident light beam is not pure-white, the resulting spectrum will exhibit colors with differing intensities, as shown in the following illustration:



We could say that this non-white light consists of a blend of colors, *but not an equally-proportionate blend*. Some colors are more strongly represented than others within a non-white light beam.

If the incident light is *monochromatic* (i.e. consists of just a single color), it will not split into a spectrum of colors when passing through a prism, but instead will emerge as a single beam of that same color, as shown in this next illustration:



In order to understand what is happening here, we need to realize that light is a form of *electromagnetic radiation* consisting of oscillating electric and magnetic fields rippling through space. A beam of monochromatic light such as the red light shown above consists of electromagnetic oscillations of a single frequency. Beams of similarly monochromatic light of different color fundamentally differ in the frequency of these oscillations: red light representing the lowest frequency within the visible light spectrum and violet light representing the highest. Prisms work by *refracting* (i.e. bending) light beams according to their frequency, with higher-frequency light beams refracting at greater angles than lower-frequency light beams.

White light, which consists of a blend of many different light-wave frequencies, separates into bands of color because each of those colors represents one of the frequencies present in the white-light mix, each of them refracted at a different angle passing through the prism according to its frequency. However, a beam of monochromatic light such as the red light beam shown above only possesses one frequency, and so there is no blend to separate: the beam refracts but does not change color or separate into other colors.

Light is therefore, quite literally, more than meets the eye. While some colors of light consist only of one frequency, others are in fact composites of multiple frequencies which may be separated and analyzed using a prism. As we shall soon see, however, the pure-versus-blended character of oscillating waves is not limited to light. There are, in fact, many types of oscillating phenomena that may consist of mixtures of different frequencies, separable into multiple "pure" frequencies.

4.2 Fourier's Theorem

This principle, of some waves being comprised of simpler waves of different frequency (and sometimes) differing intensity, is not limited to light. The exact same principle extends to all manner of waves including mechanical vibrations, sound waves, and AC electrical signals, and is often called *Fourier's Theorem* in honor of the French mathematician Jean Baptiste Joseph Fourier who developed it:

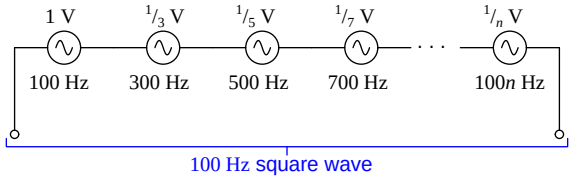
Fourier's Theorem

> **Any wave-shape, no matter how complicated, is equivalent to a sum of purely sinusoidal waveforms.**

For simple sinusoidal (i.e. monochromatic) waves, that sum contains just one term: a pure sinusoid. For non-sinusoidal waves, that sum consists of multiple sinusoidal waves of differing frequency and usually of differing amplitude as well. In some cases, the sum consists of an *infinite series* of sinusoidal waves. The important point to understand here is that *any* wave that is not a pure sinusoid itself is actually equivalent to a set of pure sinusoids added together. Although the process of determining what those exact sinusoids are (amplitude, frequency, and phase shift relative to each other) can be quite complex, we know for a fact that this equivalence is real in all cases.

If the wave in question happens to be *periodic* – that is, it repeats itself identically over some interval of time known as its *fundamental period* – then the sinusoidal frequencies comprising that wave will all have frequencies that are whole-number multiples of the frequency associated with that period. For example, if a waveform of any shape precisely repeats itself 100 times per second, then the only sinusoidal frequencies that may possibly be found in that wave will be whole-numbered multiples of 100 Hz: i.e. 100 Hz, 200 Hz, 300 Hz, 400 Hz, etc., and we refer to these as *harmonic* frequencies. The first harmonic (100 Hz in this example) is called the wave's *fundamental* frequency.

Of course, it's possible for a wave-shape to be such that it contains some harmonics but not all possible harmonics. For example, a perfect square wave with a 50% duty cycle happens to contain only odd harmonics and no even; e.g. a 100 Hz square wave consists of a 100 Hz sinusoid superimposed with sinusoids at 300 Hz, 500 Hz, 700 Hz, etc. all with amplitudes diminishing as a function of the reciprocal of the harmonic number. The following network shows how we could "build" a 100 Hz square wave by series-connecting an infinite number of sinusoidal AC voltage sources tuned to harmonic multiples of 100 Hz and with appropriately diminishing amplitudes:



Periodic waves of different shape contain either different sets of harmonic frequencies, and/or different amplitudes for each. This is a very non-intuitive yet powerful concept, that the essential "recipe" for constructing any shape of wave varies only in which harmonics we include and how much

of each. Conversely, we may regard any shape of wave as equivalent to a series of sinusoids at those appropriate frequencies and amplitudes. This is important because sinusoids are mathematically "simple" to analyze, which means any complicated wave-shape may be analyzed as a composite of simpler waves. To use the color analogy again, we may construct any color of light we might wish simply by combining the correct proportions of red, green, and/or blue (the so-called "RGB" color model), and conversely we may regard any arbitrary color as being the superposition of a certain amount of red light, a certain amount of green light, and a certain amount of blue light.

4.3 Fourier series

Fourier's Theorem states that any waveform is equivalent to a sum, or superposition, of sinusoidal waveforms. Expressing this more precisely for any given waveform requires a series of cosine and/or sine terms added together in what is referred to as a *Fourier series*:

In a Fourier series we have individual terms representing each harmonic contained in the waveform, with amplitude coefficients (a) for each. One such expression of a generic Fourier series is seen here:

$$a_0+(a_1\cos\omega t+b_1\sin\omega t)+(a_2\cos 2\omega t+b_2\sin 2\omega t)+(a_3\cos 3\omega t+b_3\sin 3\omega t)+\cdots(a_n\cos n\omega t+b_n\sin n\omega t)$$

Where,
a = Peak amplitude (for each harmonic), any unit
ω = Frequency (for each harmonic), in radians per second
t = Time, in seconds

In any Fourier series a_0 is the DC component of the waveform (i.e. the "zeroth" harmonic) while $(a_1\cos\omega t + b_1\sin\omega t)$ are a cosine and sine function-pair representing the first harmonic (i.e. the fundamental). Other harmonics have their own amplitude coefficients and frequencies, the above formula ending with the nth harmonic which may be arbitrary or even infinite in order.

For example, below we see the Fourier series for a perfect square wave, which happens to be infinite:

$$\sin\omega t + \frac{1}{3}\sin 3\omega t + \frac{1}{5}\sin 5\omega t + \frac{1}{7}\sin 7\omega t + \cdots + \frac{1}{n}\sin n\omega t$$

This mathematical series may be thought of as a "recipe" for building a square wave. It tells us that if you take a sine wave of any given frequency, then add to that another sine wave one-third as tall at three times the frequency, then add to that another sine wave one-fifth as tall as the original at five times the frequency, and so on to infinity, you will get a perfect square wave. Note the absence of any cosine terms, as well as the absence of a DC (a_0) term. The generic form of the Fourier series seen earlier contains all possible terms for any type of waveform, and so we should not be surprised if the Fourier series for any *particular* wave-shape may be missing some of these.

The cosine-sine form of the Fourier series first introduced in this section is not the only way to generically represent a Fourier series. All a Fourier series needs to do is represent the amplitude and phase angle of each harmonic term, and a cosine-sine function pair is just one valid method of doing so[1]. An alternative Fourier series could be written using nothing but sine terms with phase offsets for each harmonic:

$$A_0 + A_1 \sin(\omega t + \theta_1) + A_2 \sin(2\omega t + \theta_2) + A_3 \sin(3\omega t + \theta_3) + \cdots A_n \sin(n\omega t + \theta_n)$$

Where,
A = Peak amplitude (for each harmonic), any unit
ω = Frequency (for each harmonic), in radians per second
t = Time, in seconds
θ = Phase shift (for each harmonic), in radians

It's even possible to write Fourier series using exponential notation, based on Euler's Relation where $e^{jx} = \cos x + j \sin x$:

$$A_0 + A_1 e^{j(\omega t + \theta_1)} + A_2 e^{j(2\omega t + \theta_2)} + A_3 e^{j(3\omega t + \theta_3)} + \cdots A_n e^{j(n\omega t + \theta_n)}$$

Where,
A = Peak amplitude (for each harmonic), any unit
j = Imaginary operator = $\sqrt{-1}$
ω = Frequency (for each harmonic), in radians per second
t = Time, in seconds
θ = Phase shift (for each harmonic), in radians

Regardless of how we might choose to write a Fourier series, the basic principle at work is Fourier's Theorem – that we may synthesize any arbitrary waveform by summing together enough sinusoidal waves at different amplitudes, frequencies, and phase shifts. Any mathematical expression of sinusoidal functions necessary to synthesize a particular wave-shape is called the Fourier series, constituting a "recipe" for building that particular wave-shape.

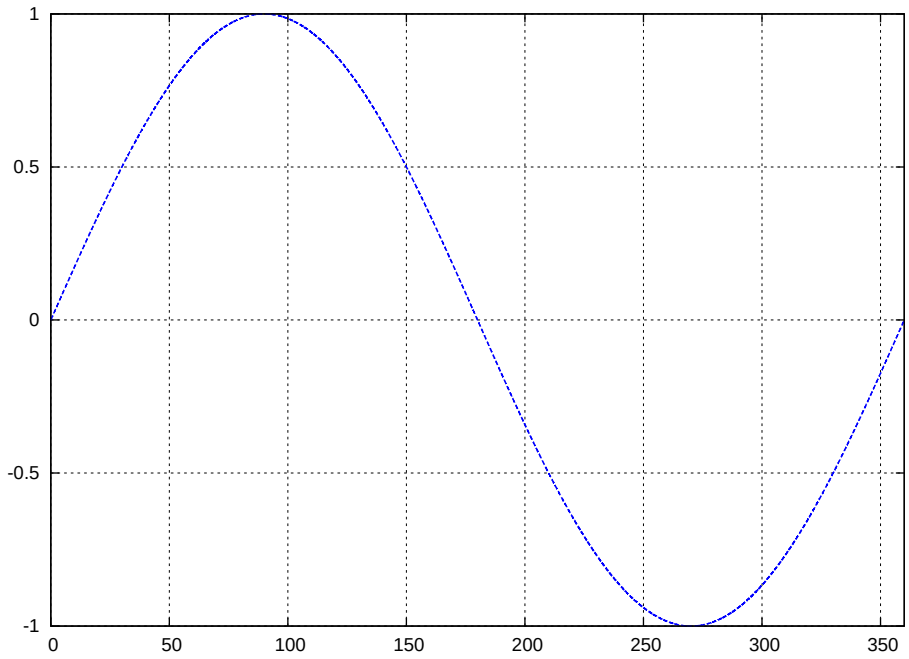
[1] If we view a right-triangle's hypotenuse as being the vector sum of its adjacent (sine) and opposite (cosine) vectors, it should be clear that we may represent any vector length and angle simply by summing the appropriate cosine and sine terms.

4.4 Building a square wave from sine waves

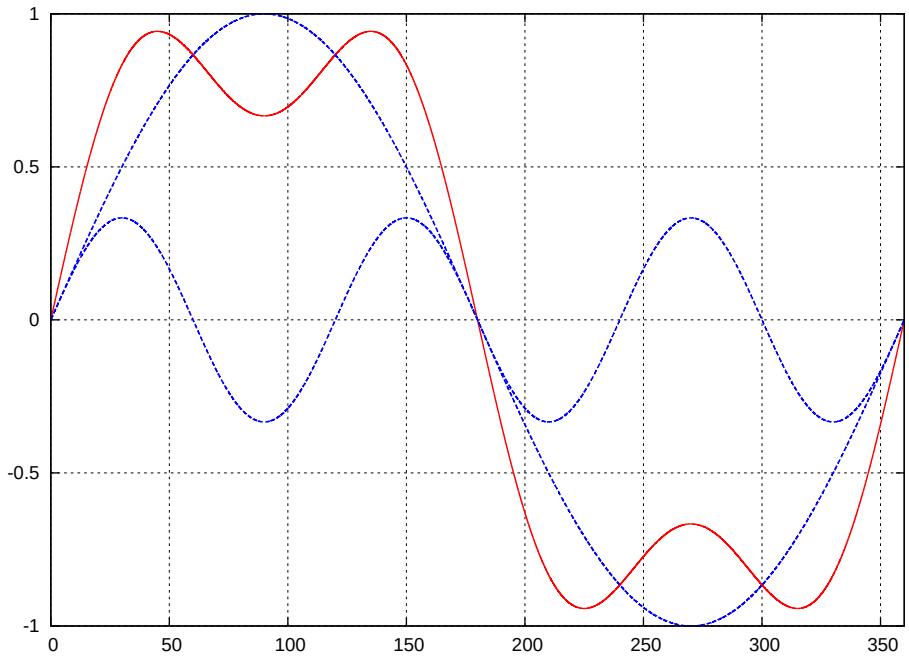
Fourier's Theorem – the claim that any non-sinusoidal waveform is equivalent to a series of superimposed sinusoidal waves – is a difficult one to accept. Even some of Jean Baptiste Joseph Fourier's mathematical contemporaries were skeptical. This skepticism is especially understandable when we consider wave-shapes such as square waves, which by definition have sharp-edged shapes in the time domain. It seems quite baffling that any wave-shape with square corners might somehow be comprised of smooth-shaped waves added together over time.

What follows in this section is not a proof of Fourier's central claim, but merely an illustration of how we can indeed produce a sharp-edged wave simply by superimposing sinusoidal waves. To do this, we will follow the Fourier Series for a square wave, which is an infinite series of sine waves added together, these sine waves' frequencies being integer-multiples of the square wave's fundamental frequency, and their magnitudes following a carefully prescribed proportionality. Please note that the particular mathematical "recipe" we are about to follow works specifically to create square waves, and that a different "recipe" (i.e. different harmonic numbers and/or different magnitudes of these harmonics) will result in some wave-shape other than a square wave.

First, we will begin with a blue-colored plot of one sine wave over a domain of 0 to 360 degrees, oscillating between peak values of $+1$ and -1:



Next, we will plot that same blue-colored sine wave as well as another blue-colored sine wave at three times' higher frequency and one-third magnitude, and plot their *sum* in red:

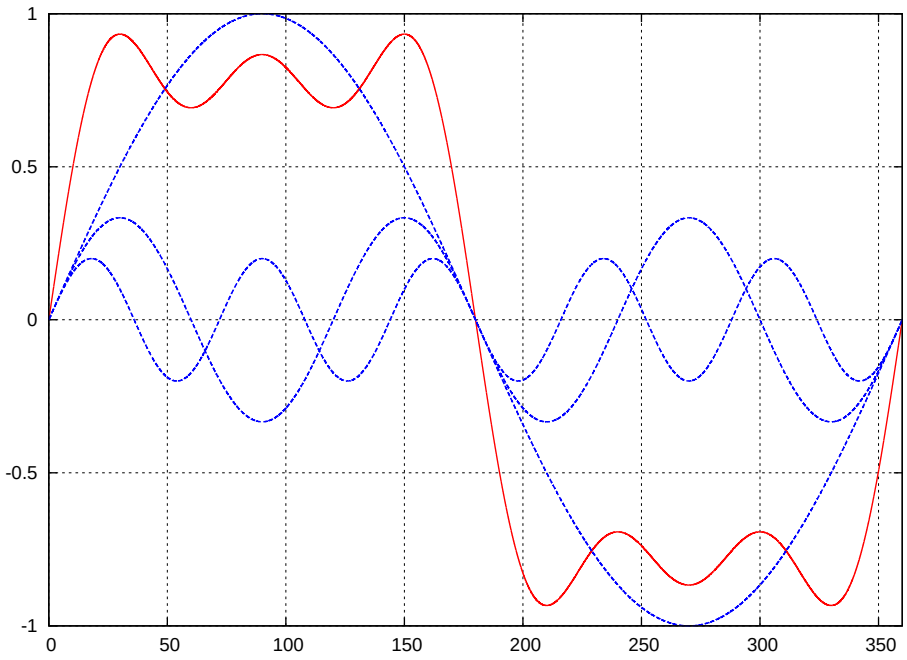


Here, the low-frequency blue sine wave is the *fundamental* or *first harmonic*, while the high-frequency blue sine wave is the *third harmonic* due to its three-times higher frequency. The fact that we chose to make this third harmonic have one-third the magnitude of the fundamental is because we are following the Fourier Series (i.e. the "recipe") for a square wave which calls for each successive harmonic to be proportionately smaller in magnitude:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \cdots + \frac{1}{n} \sin n\omega t$$

We will continue to plot and sum odd-numbered harmonics, each one at a proportionately-reduced magnitude, in order to show how the accumulation of these harmonics at the ratios specified by this particular Fourier Series "recipe" end up producing an approximation of a square wave.

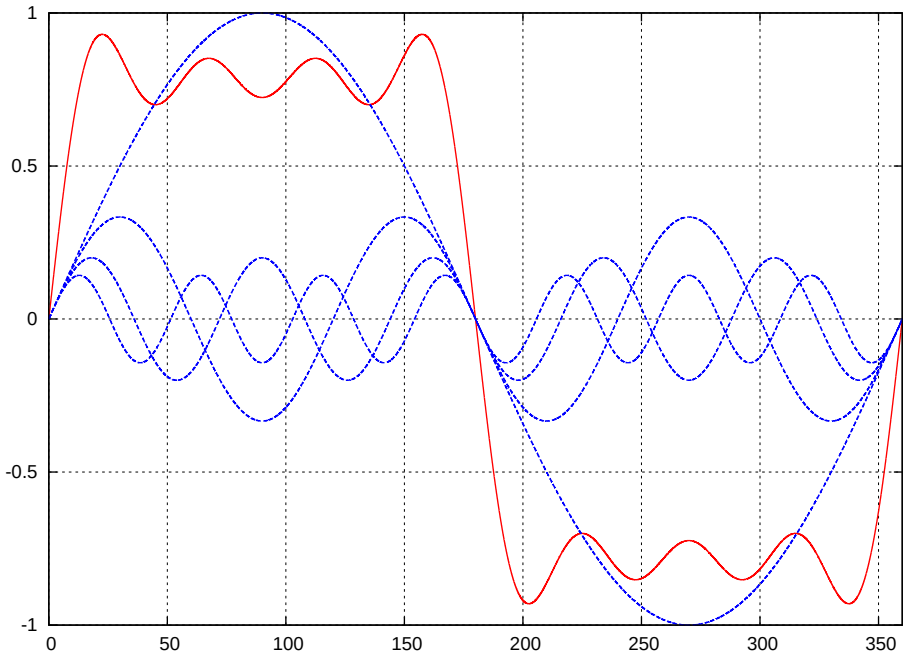
Next, we will plot that same blue-colored sine wave as well as another blue-colored sine wave at three times' higher frequency and one-third magnitude as well as another blue-colored sine wave at five times' higher frequency and one-fifth magnitude, and plot their *sum* in red:



At this point, the red-colored waveform is described by the following mathematical series, with first, third, and fifth terms of the ideal square-wave Fourier Series:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t$$

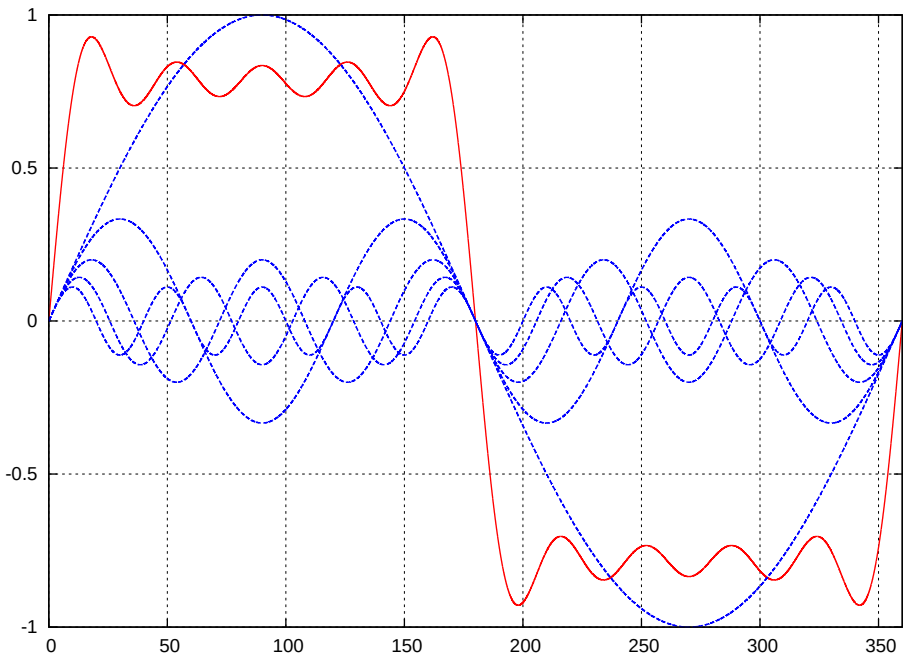
Next, we will add a seventh harmonic to the plot:



At this point, the red-colored waveform is described by the following mathematical series, with first, third, fifth, and seventh terms of the ideal square-wave Fourier Series:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \frac{1}{7} \sin 7\omega t$$

Next, we will add a ninth harmonic to the plot:



At this point, the red-colored waveform is described by the following mathematical series, with first, third, fifth, seventh, and ninth terms of the ideal square-wave Fourier Series:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \frac{1}{7} \sin 7\omega t + \frac{1}{9} \sin 9\omega t$$

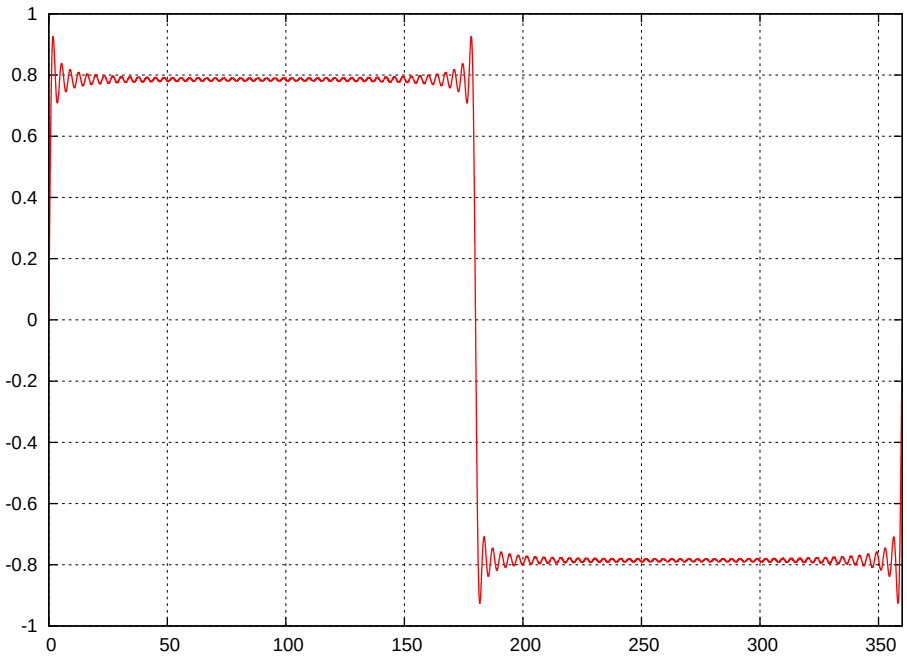
By now hopefully the pattern is clear to see: with each successive odd-numbered harmonic added to the sum, the superposition of these sine waves comes closer and closer to resembling a true square wave. For all future simulations I will plot only the sum (in red) so as to avoid the clutter that would otherwise result from all the sinusoids plotted on top of each other.

Here we see the sum of all odd-numbered harmonics up to the fifteenth, following the same inverse-proportional magnitude pattern as before:

Harmonic series simulated:

$$\sin\omega t + \frac{1}{3}\sin 3\omega t + \frac{1}{5}\sin 5\omega t + \frac{1}{7}\sin 7\omega t + \frac{1}{9}\sin 9\omega t + \frac{1}{11}\sin 11\omega t + \frac{1}{13}\sin 13\omega t + \frac{1}{15}\sin 15\omega t$$

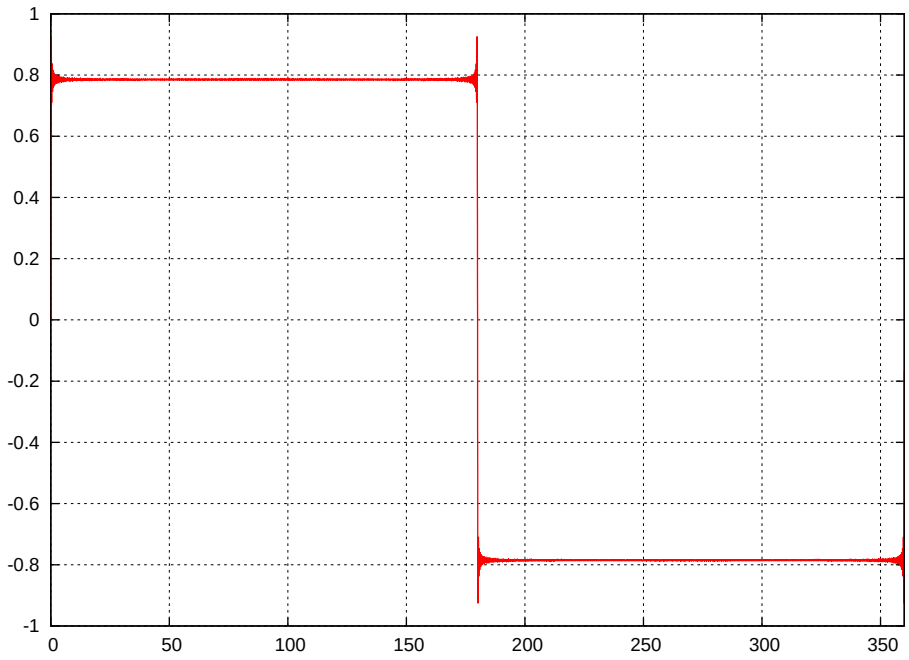
Here we see the sum of all odd-numbered harmonics up to the ninety-ninth, following the same inverse-proportional magnitude pattern as before:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \cdots + \frac{1}{99} \sin 99\omega t$$

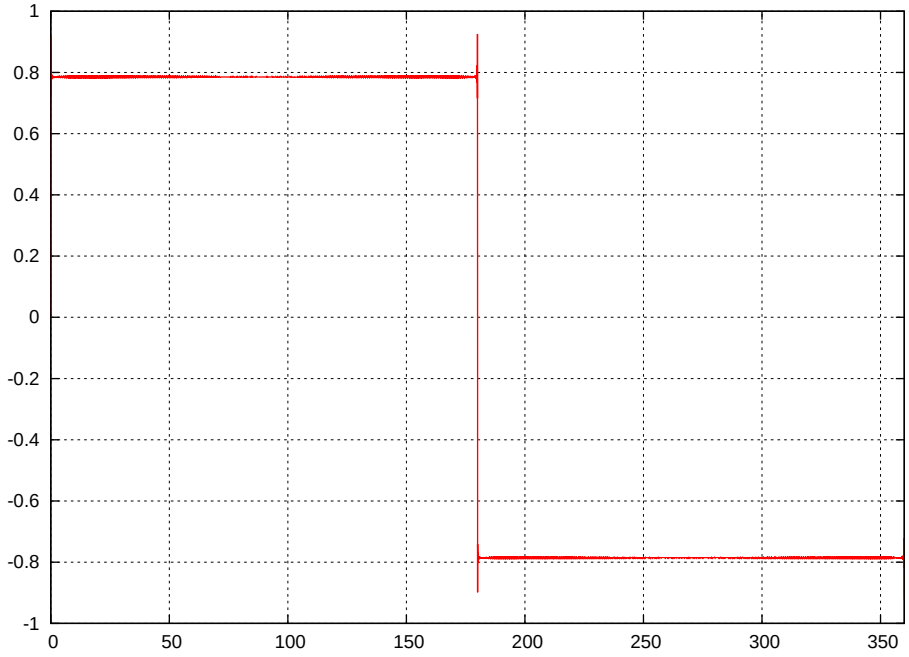
Here we see the sum of all odd-numbered harmonics up to the 999th, following the same inverse-proportional magnitude pattern as before:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \cdots + \frac{1}{999} \sin 999\omega t$$

Here we see the sum of all odd-numbered harmonics up to the 9999th, following the same inverse-proportional magnitude pattern as before:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \cdots + \frac{1}{9999} \sin 9999\omega t$$

As you can see, it is indeed possible to very closely simulate a perfect square wave by summing together a long series of sine waves having just the right harmonic frequencies and magnitudes. However, it is important to understand that the Fourier Series for a square wave is not merely a "recipe" for synthesizing a square wave from a multitude of sinusoidal wave sources, but is really a *mathematical equivalence* between these two things. In other words, it is equally true that any square-wave oscillation actually contains all those sinusoidal frequencies within it even though the creation of that square-wave oscillation may have had nothing to do with sinusoidal causes.

For example, if we were to built a digital electronic oscillator circuit producing an voltage signal pulsing in a square-wave manner, the final output stage of this circuit being a simple pair of transistors alternately turning on and off, and use this oscillator's output signal to energize some other network of components, that network of components would react to the square-wave voltage in precisely the same manner as it would if energized by a multitude of sine-wave voltage sources (at the frequencies and magnitudes prescribed by the square-wave Fourier Series) all wired together in series! In other words, that network could not "tell the difference" between the infinite array of sine-wave voltage sources versus the simple square-wave digital oscillator – the Fourier Series for a square wave being mathematically the same thing as the real square wave.

This equivalence between one wave-shape and a collection of sinusoidal waves can have profound consequences in electronics. One of these consequences is radio interference caused by high-frequency digital switching circuits. Imagine if you will a digital circuit operating with a square-wave oscillator frequency of 40 MHz. If this circuit is energized outside of a shielded metal enclosure, there will be some inevitable radiation of electromagnetic waves from the circuit into the surrounding space. This radiation can cause interference with nearby radio-communication systems which also use electromagnetic waves to transmit and receive information. Obviously, if a nearby radio system were designed to operate using 40 MHz signals, the emissions from this digital circuit with its 40 MHz oscillator signal could interfere with those radio communications. However, the fact that the digital circuit's oscillator produces a *square* wave means the radiated signal is actually equivalent to 40 MHz, 120 MHz, 200 MHz, 280 MHz, 360 MHz, and higher (odd-harmonic) sinusoidal frequencies being broadcast simultaneously, potentially interfering with any radio communication system(s) operating at any of those frequencies! This means a signal such as a square wave with its abundance of harmonics poses a more severe threat of radio interference than, say, a pure sine wave oscillator operating at 40 MHz.

4.5 Building a triangle wave from cosine waves

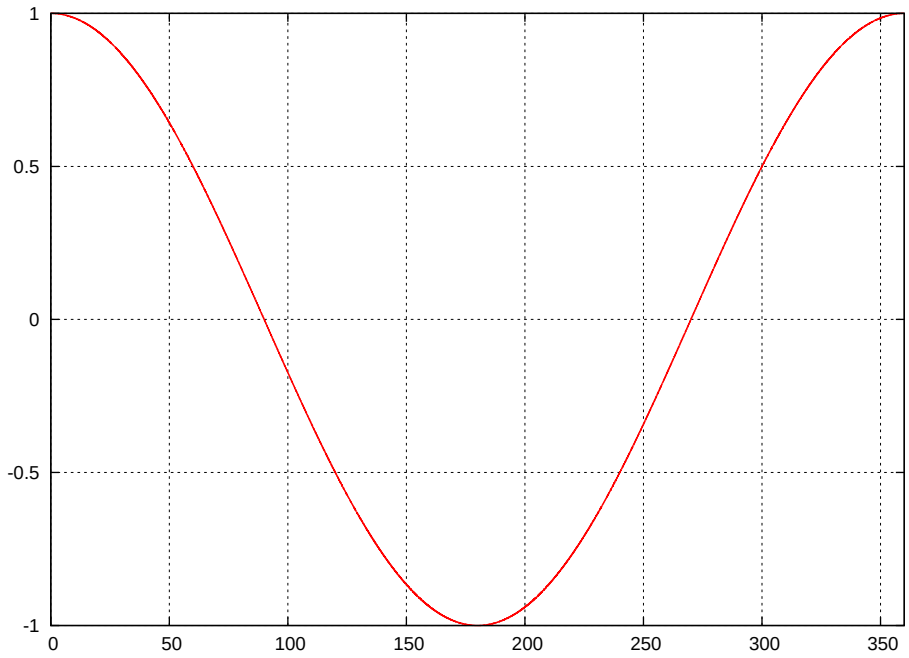
The Fourier Series for a triangle wave is different from that of a square wave. Square waves consist of an infinite sum of sine waves at odd harmonics, each harmonic's magnitude being $\frac{1}{n}$ of the fundamental's:

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \frac{1}{7} \sin 7\omega t + \cdots + \frac{1}{n} \sin n\omega t$$

Triangle waves, by contrast, are also an infinite sum of odd harmonics, but these are cosine waves rather than sine waves, and the magnitude ratios are different. Instead of each successive harmonic having a magnitude of $\frac{1}{n}$ compared to the fundamental, the ratio is $\frac{1}{n^2}$. This, in essence, constitutes a different "recipe" for a non-sinusoidal wave, but still comprised of the same basic "ingredients" (i.e. sinusoidal waves at harmonic frequencies) as the square wave:

$$\cos \omega t + \frac{1}{9} \cos 3\omega t + \frac{1}{25} \cos 5\omega t + \frac{1}{49} \cos 7\omega t + \cdots + \frac{1}{n^2} \cos n\omega t$$

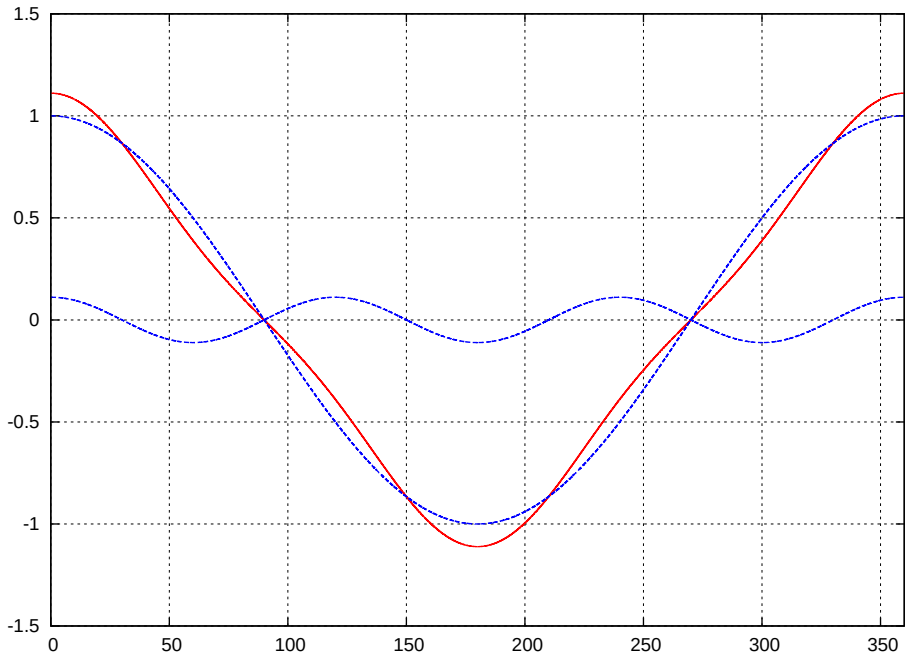
As with the square wave synthesis example in the previous section, we will build up this new waveform harmonic by harmonic. Here we see the fundamental (i.e. first harmonic) plotted on its own:



Harmonic series simulated:

$$\cos \omega t$$

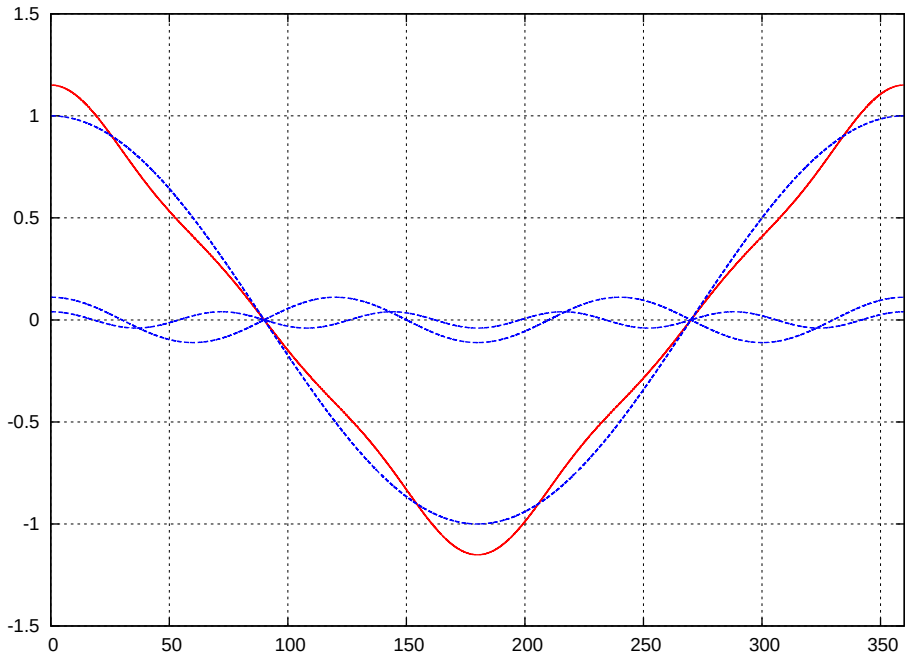
Next we see the first and third harmonics in blue as separate cosine waves, as well as their sum shown in red:



Harmonic series simulated:

$$\cos \omega t + \frac{1}{9} \cos 3\omega t$$

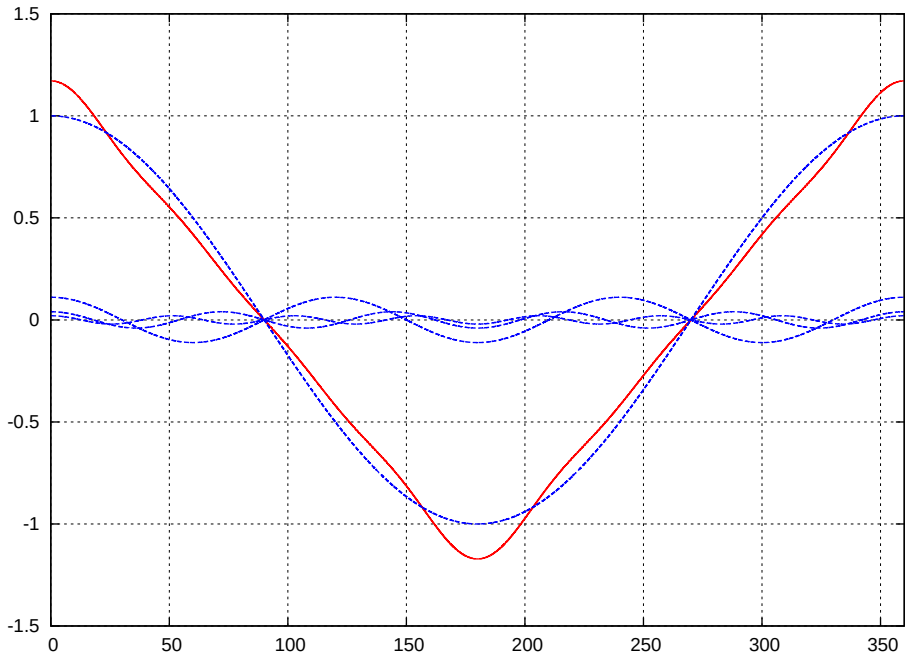
Here we have the first, third, and fifth harmonics plotted as separate cosine waves in blue, as well as their summation shown in red:



Harmonic series simulated:

$$\cos \omega t + \frac{1}{9} \cos 3\omega t + \frac{1}{25} \cos 5\omega t$$

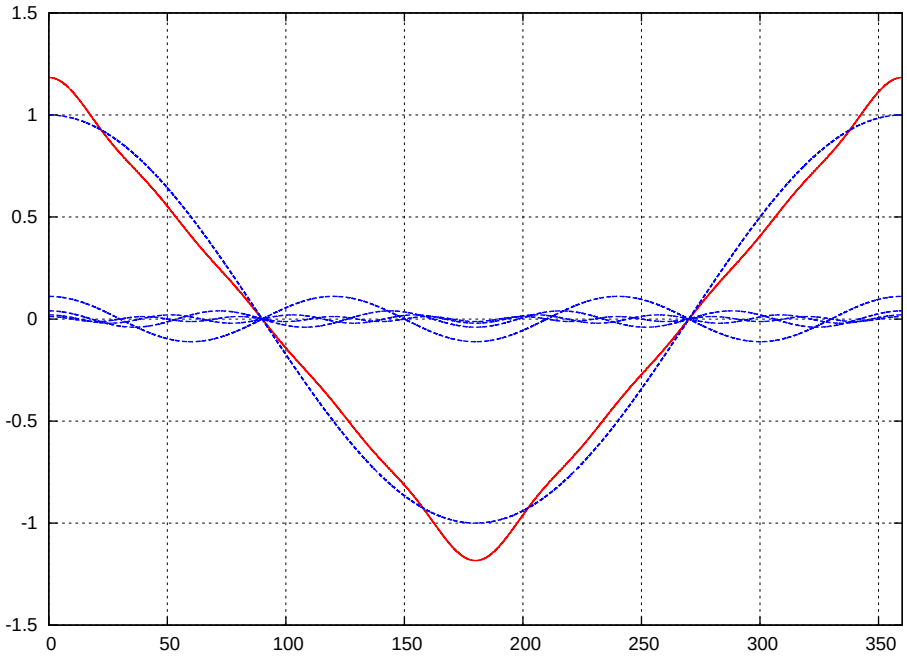
Adding the seventh harmonic to the mix, we see the individual harmonics plotted in blue along with the sum plotted in red:



Harmonic series simulated:

$$\cos\omega t + \frac{1}{9}\cos 3\omega t + \frac{1}{25}\cos 5\omega t + \frac{1}{49}\cos 7\omega t$$

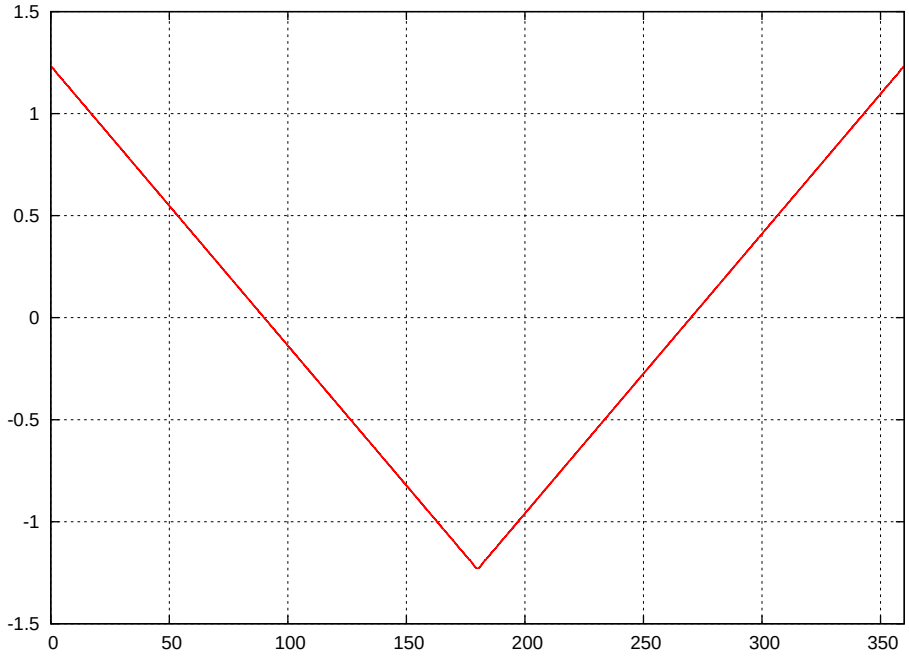
Continuing our progression with the addition of the ninth harmonic:



Harmonic series simulated:

$$\cos\omega t + \frac{1}{9}\cos 3\omega t + \frac{1}{25}\cos 5\omega t + \frac{1}{49}\cos 7\omega t + \frac{1}{81}\cos 9\omega t$$

With every additional odd harmonic (proportioned according to the inverse-square of the harmonic number), the summation waveform clearly takes on the shape of a triangle wave. Extending the series of odd-harmonic cosine terms all the way to the 99th sharpens this approximation until it appears nearly perfect to our visual inspection:



Harmonic series simulated:

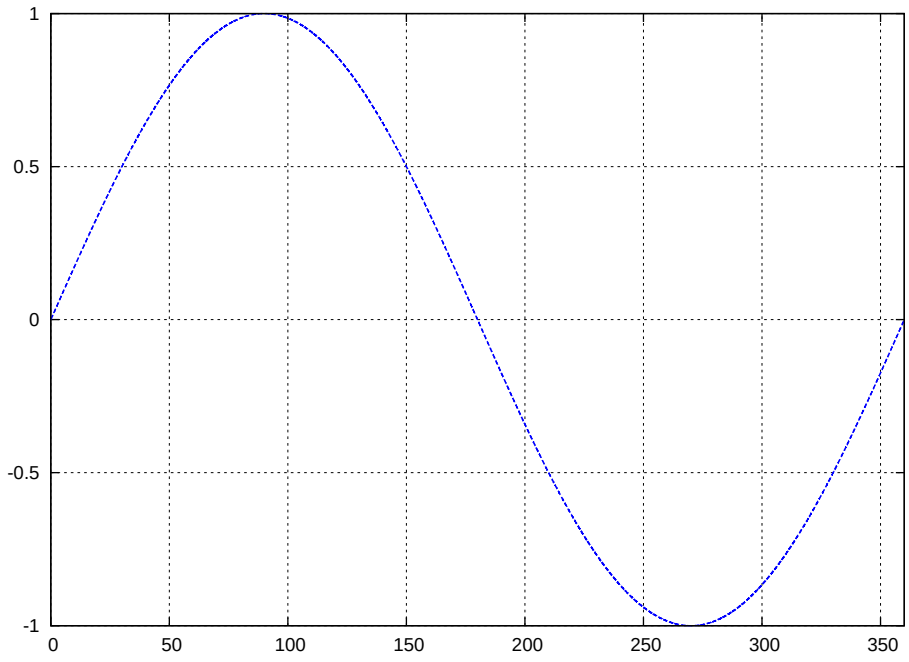
$$\cos\omega t + \frac{1}{9}\cos 3\omega t + \frac{1}{25}\cos 5\omega t + \cdots + \frac{1}{99^2}\cos 99\omega t$$

4.6 Building a sawtooth wave from sine waves

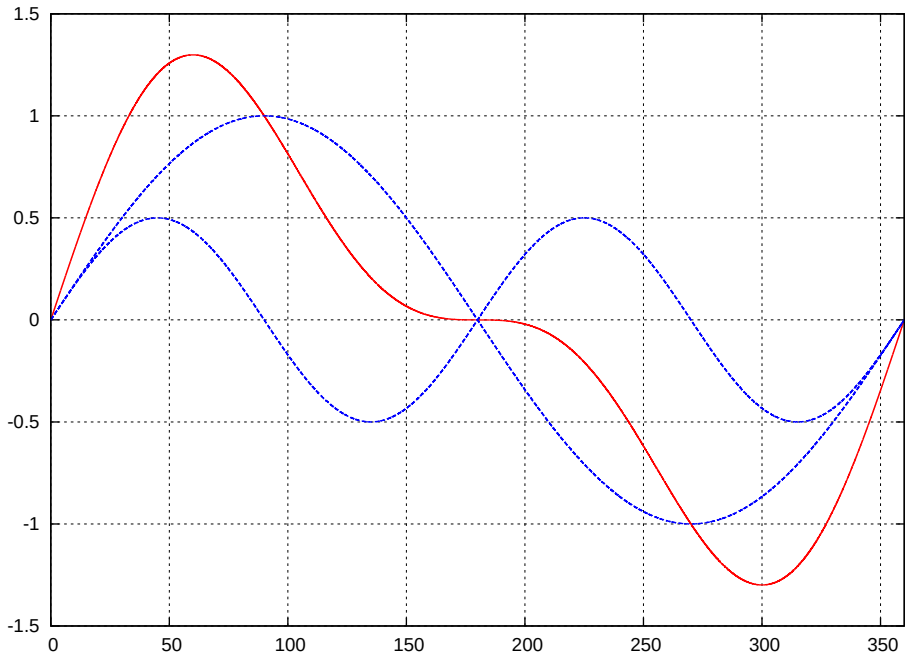
Sawtooth waves are asymmetrical about the time axis, requiring even-numbered as well as odd-numbered harmonics in the Fourier Series for this wave-shape. Here we see the infinite Fourier Series for a sawtooth wave:

$$\sin \omega t + \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{4} \sin 4\omega t + \cdots + \frac{1}{n} \sin n\omega t$$

To begin our harmonic-by-harmonic synthesis of a sawtooth wave, we will start as we did for the square wave synthesis: a blue-colored plot of one sine wave over a domain of 0 to 360 degrees, oscillating between peak values of +1 and −1:



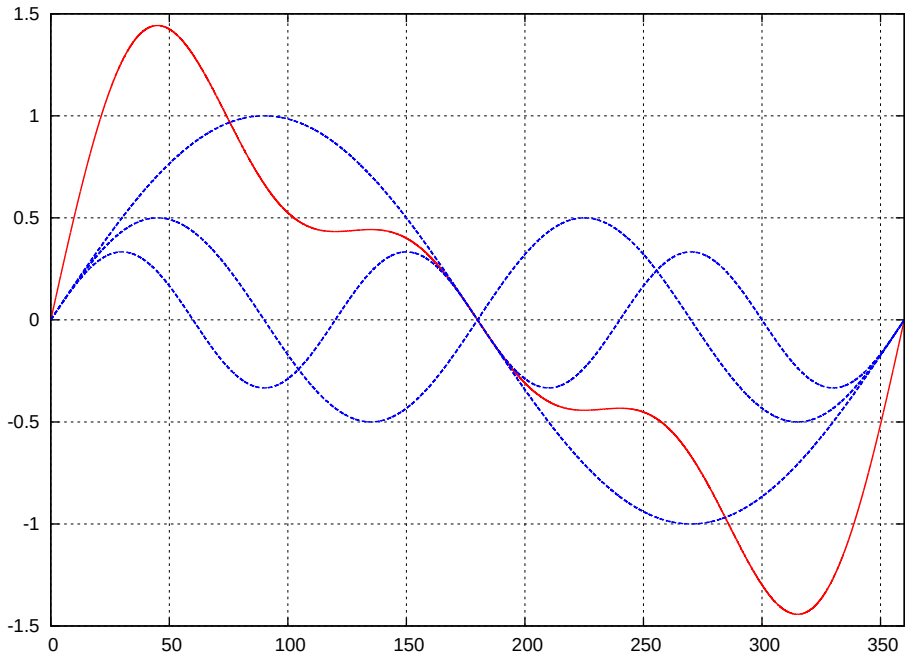
Next we will add the second harmonic at one-half the magnitude of the fundamental, showing the first and second harmonic sine waves in blue with the summation in red:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{2} \sin 2\omega t$$

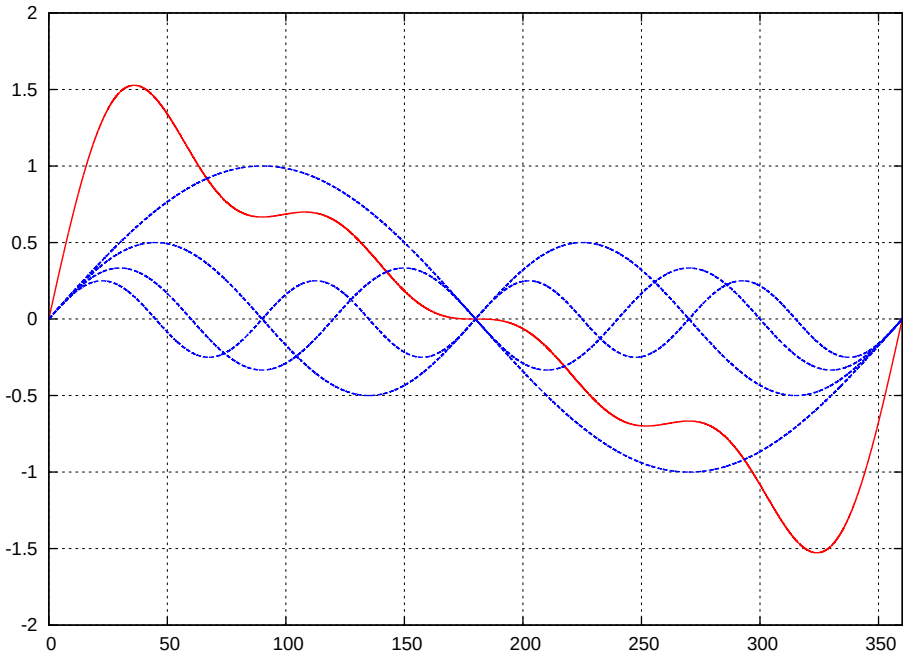
Next we will add the third harmonic at one-third the magnitude of the fundamental, showing the first three harmonic sine waves in blue with the summation in red:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t$$

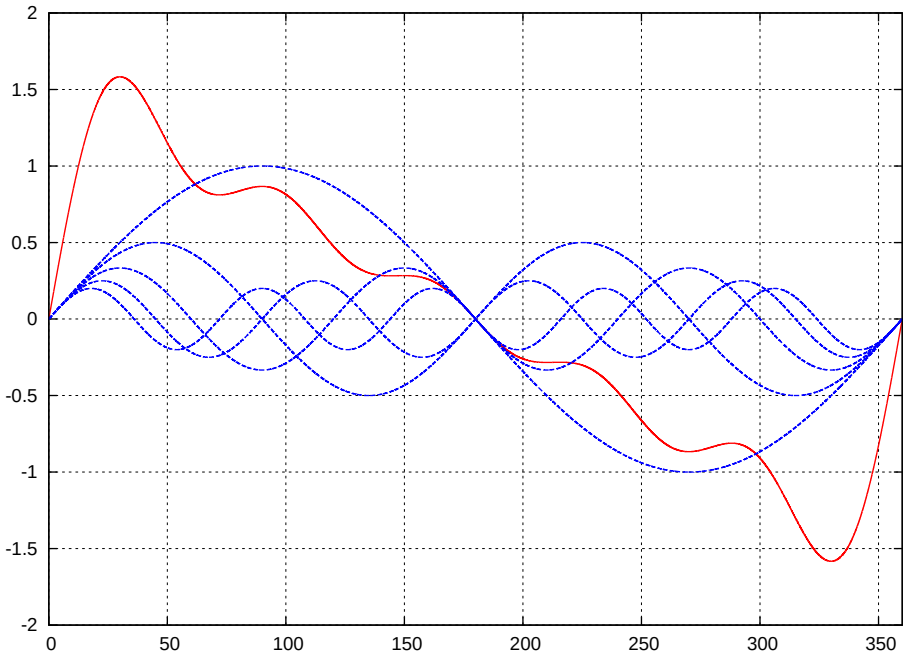
Adding the fourth harmonic:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{4} \sin 4\omega t$$

Adding the fifth harmonic:

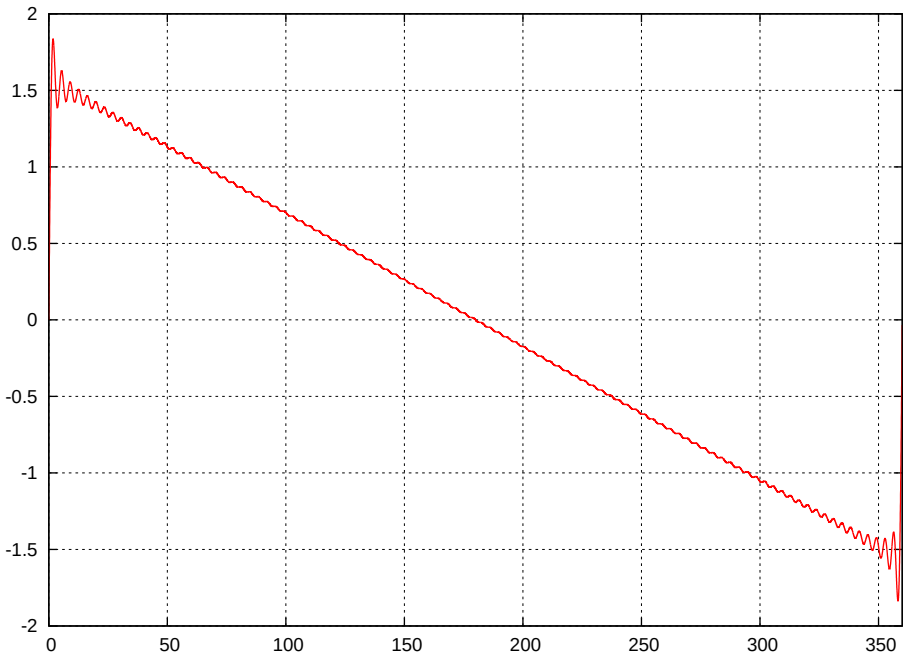


Harmonic series simulated:

$$\sin \omega t + \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{4} \sin 4\omega t + \frac{1}{5} \sin 5\omega t$$

It should be clear by now the evolutionary progression of this wave-shape, gradually becoming more linear in its downward slope and more vertical in its upward.

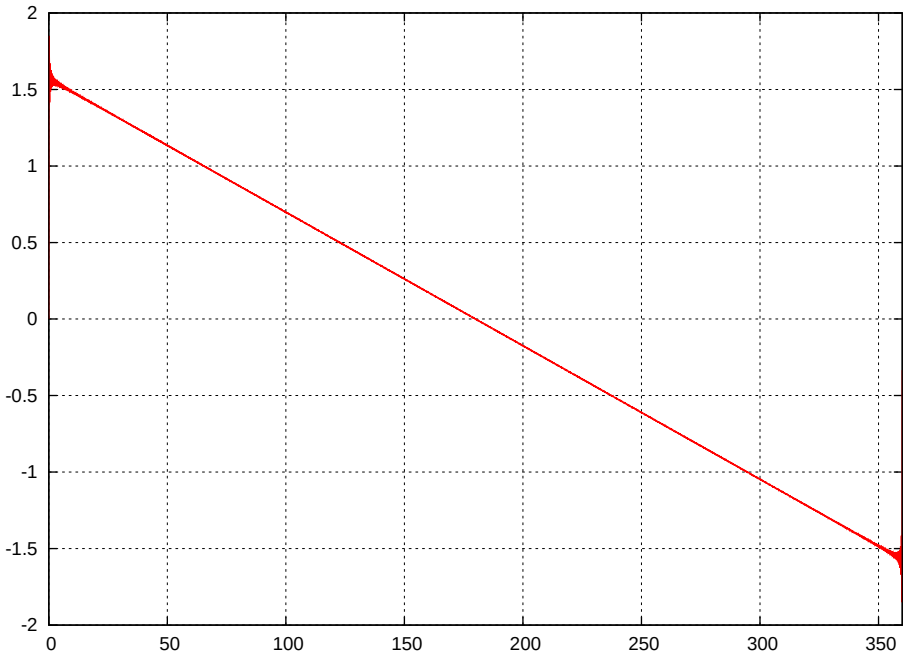
Plotting the summation of all harmonics through the 99th, omitting the individual (blue-colored) harmonic sine waves from the plot:



Harmonic series simulated:

$$\sin \omega t + \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{4} \sin 4\omega t + \frac{1}{5} \sin 5\omega t + \cdots + \frac{1}{99} \sin 99\omega t$$

Plotting the summation of all harmonics through the 999th:



Harmonic series simulated:

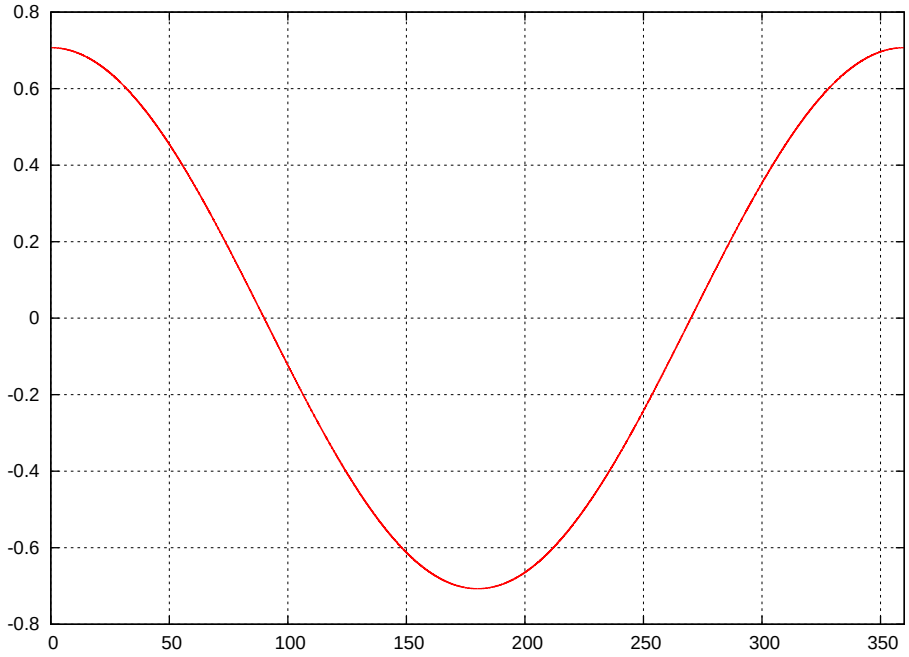
$$\sin \omega t + \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{4} \sin 4\omega t + \frac{1}{5} \sin 5\omega t + \cdots + \frac{1}{999} \sin 999\omega t$$

4.7 Building a pulse wave from cosine waves

A rectangular-shaped pulse waveform with a duty cycle (D) other than 50% is commonly referred to as a *pulse* wave. Plain square waves are just special cases of pulse waves. Pulse waves consist of an infinite sum of cosine waves at both even and odd harmonics, each harmonic's magnitude being $\frac{1}{n}$ of the fundamental's multiplied by another term that is a sine function of the duty cycle ($\sin(n\pi D)$) where $n\pi D$ is in *radians* rather than degrees:

$$\sin(\pi D)\cos(\omega t) + \frac{1}{2}\sin(2\pi D)\cos(2\omega t) + \frac{1}{3}\sin(3\pi D)\cos(3\omega t) + \cdots + \frac{1}{n}\sin(n\pi D)\cos(n\omega t)$$

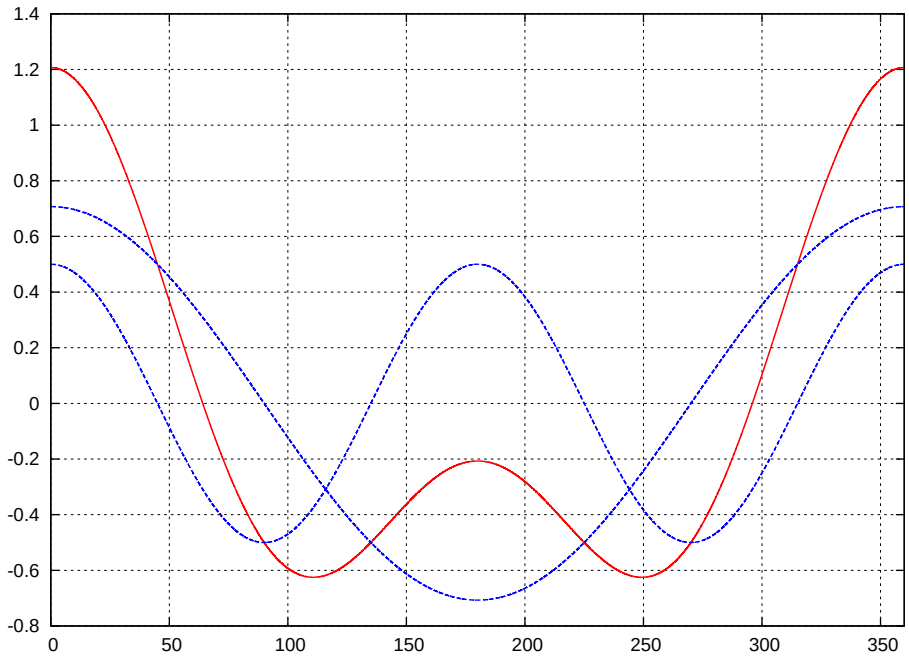
As with the wave synthesis examples in the previous sections, we will build up a pulse waveform harmonic by harmonic. For this example we will set the duty cycle to 25% ($D = 0.25$). Here we see the fundamental (i.e. first harmonic) plotted on its own:



Harmonic series simulated:

$$\sin(0.25\pi)\cos(\omega t)$$

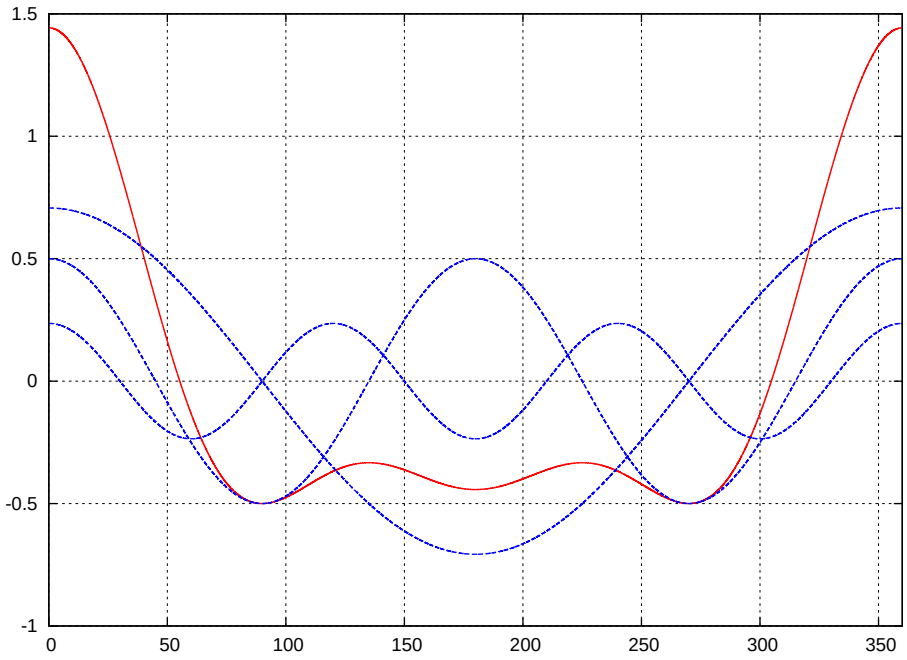
Next we see the first and second harmonics in blue as separate cosine waves, as well as their sum shown in red:



Harmonic series simulated:

$$\sin(0.25\pi)\cos(\omega t) + \frac{1}{2}\sin(2 \times 0.25\pi)\cos(2\omega t)$$

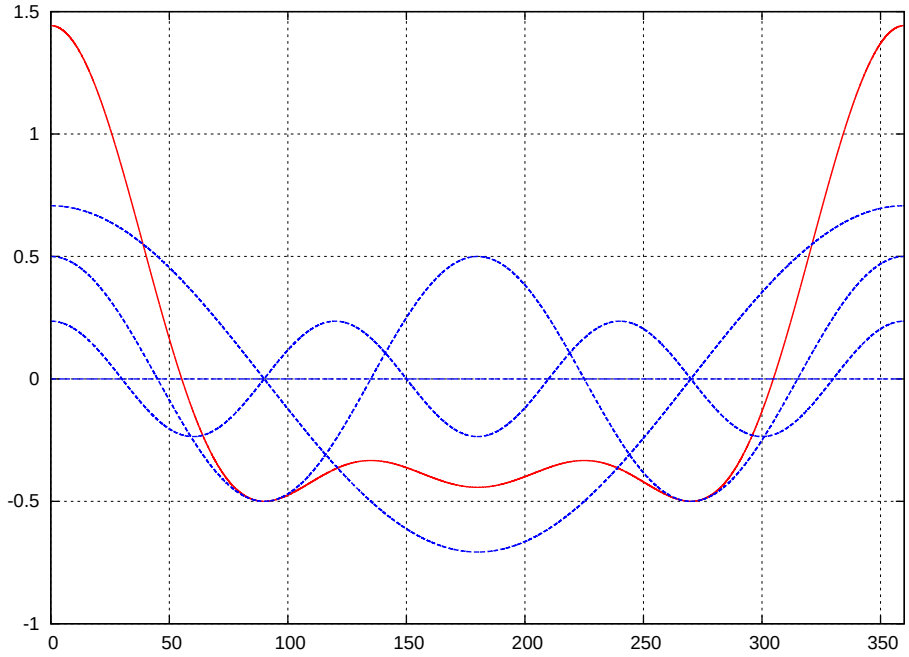
Next we see the first, second, and third harmonics in blue as separate cosine waves, as well as their sum shown in red:



Harmonic series simulated:

$$\sin(0.25\pi)\cos(\omega t) + \frac{1}{2}\sin(2 \times 0.25\pi)\cos(2\omega t) + \frac{1}{3}\sin(3 \times 0.25\pi)\cos(3\omega t)$$

Adding the fourth harmonic to the others is rather non-eventful, because the fourth harmonic's sine term becomes $\sin(4 \times 0.25\pi)$ which is the sine of π radians, equal to zero. Thus, the fourth harmonic for this $D = 0.25$ pulse waveform happens to be a flat line, contributing nothing to the sum (red wave):



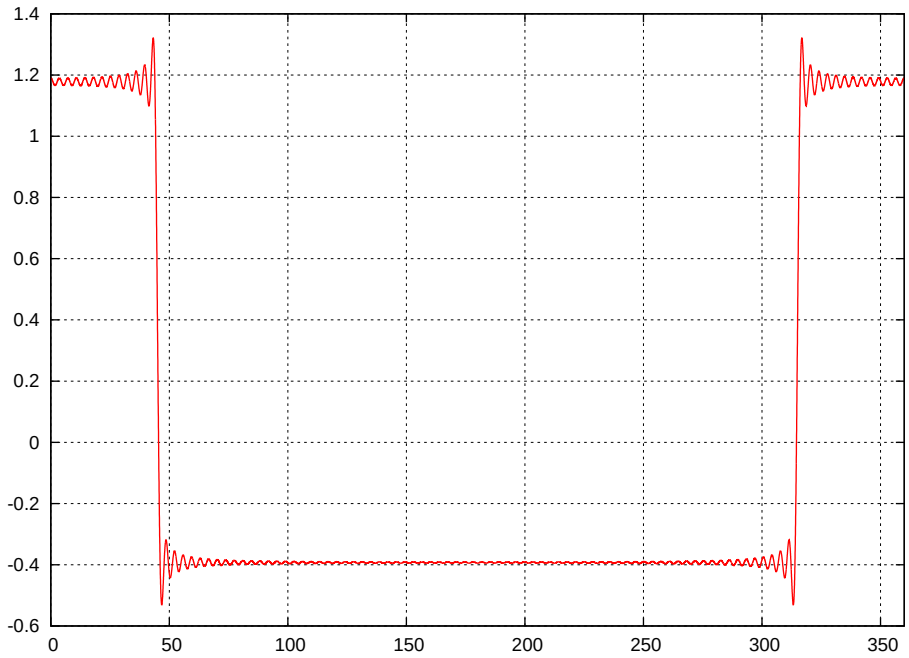
Harmonic series simulated:

$$\sin(0.25\pi)\cos(\omega t) + \frac{1}{2}\sin(2 \times 0.25\pi)\cos(2\omega t) + \frac{1}{3}\sin(3 \times 0.25\pi)\cos(3\omega t) + \frac{1}{4}\sin(4 \times 0.25\pi)\cos(4\omega t)$$

Showing the fourth harmonic with its magnitude of zero:

$$\sin(0.25\pi)\cos(\omega t) + \frac{1}{2}\sin(2 \times 0.25\pi)\cos(2\omega t) + \frac{1}{3}\sin(3 \times 0.25\pi)\cos(3\omega t) + 0$$

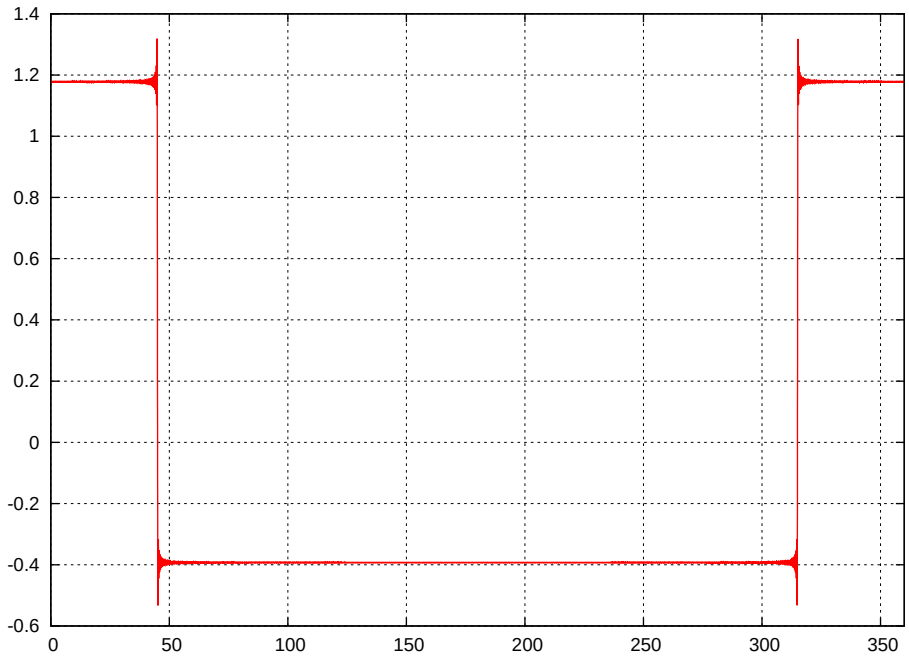
Plotting the summation of all harmonics through the 99th, omitting the individual (blue-colored) harmonic cosine waves from the plot:



Harmonic series simulated:

$$\sin(0.25\pi)\cos(\omega t) + \frac{1}{2}\sin(2 \times 0.25\pi)\cos(2\omega t) + \cdots + \frac{1}{99}\sin(99 \times 0.25\pi)\cos(99\omega t)$$

Plotting the summation of all harmonics through the 999th, omitting the individual (blue-colored) harmonic cosine waves from the plot:



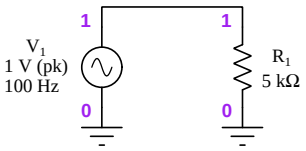
Harmonic series simulated:

$$\sin(0.25\pi)\cos(\omega t) + \frac{1}{2}\sin(2 \times 0.25\pi)\cos(2\omega t) + \cdots + \frac{1}{999}\sin(999 \times 0.25\pi)\cos(999\omega t)$$

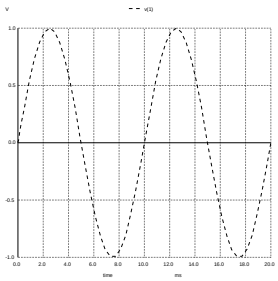
The 25% duty cycle is fairly clear to see in this plot, with the wave's period being 360 degrees and the "on" or "high" duration being 90 degrees in total (from 0 to approximately 45 degrees, and again from approximately 315 to 360 degrees).

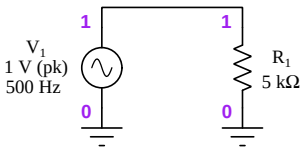
4.8 Time vs. frequency in circuits

With AC electrical signals we are able to view the shape of waves using an instrument called an *oscilloscope*, displaying voltage as a *time-domain* graph with voltage on the vertical axis and time on the horizontal. Here is a small collection of oscillographs generated using the circuit-simulation software NGSPICE, each accompanied by a schematic diagram and the "netlist" source code defining the circuit and the analysis for the software:

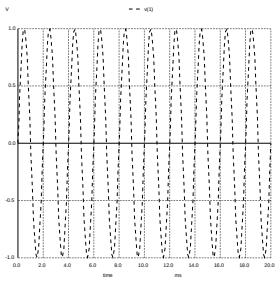


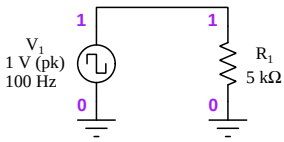

```
* Sine wave simulation
v1 1 0 sin(0 1 100 0 0)
rload 1 0 5000
.tran 0.5m 20m
.plot tran v(1)
.end
```



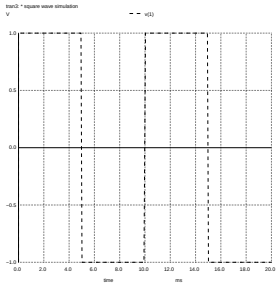



```
* Sine wave simulation
v1 1 0 sin(0 1 500 0 0)
rload 1 0 5000
.tran 0.1m 20m
.plot tran v(1)
.end
```






```
* Square wave simulation
v1 1 0 pulse(-1 1 0
+ 1n 1n 4.999m 10m)
rload 1 0 5000
.tran 0.1m 20m
.plot tran v(1)
.end
```

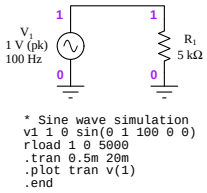


The first two waves are *sinusoidal*, which means they have the same shape as a sine (or cosine) wave. The third wave is definitely *not* sinusoidal, and is commonly known as a *square wave*.

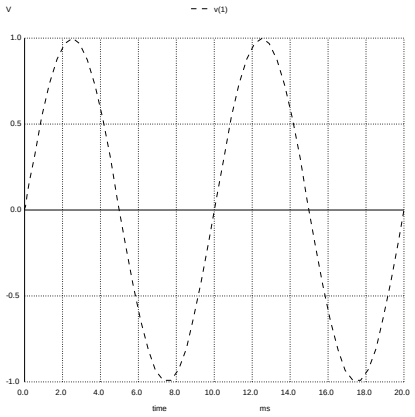
By the phrase *time-domain* we mean to say that these signals are expressed as amplitudes in relation to time. Mathematically they would be some function with time as the independent variable, for example $f(t) = A \sin \omega t$ for the sine waves where A is the peak amplitude and ω is the frequency in radians per second, or $f(t) = A \sin 2\pi f t$ where f is the frequency in cycles per second (Hz).

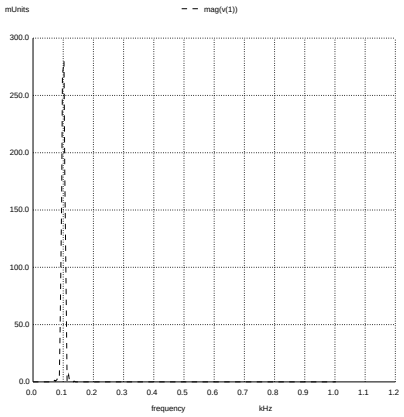
Spectrum analyzers, in contrast to oscilloscopes, display voltage as a *frequency-domain* graph, where voltage is the vertical axis and frequency is the horizontal. A spectrum analyzer is to an electrical signal as a prism is to a beam of light: each device separates a superposition of frequencies to make each component frequency clearly visible. We will explore the signal spectrum as it would be displayed by a spectrum analyzer for each of the previous four signals, once again using NGSPICE[2] to simulate the circuit and the measuring instrument.

First we will explore the slow sine wave, defined in the SPICE simulation as having a peak value of 1 Volt and a frequency of 100 Hz. Note how the spectrum for this signal consists of a single peak located at the 0.1 kHz mark on the frequency axis:

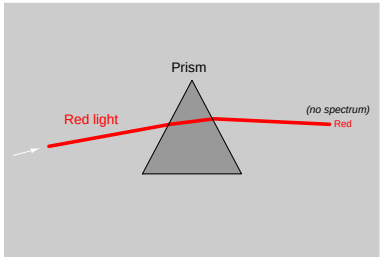



```
* Sine wave simulation
v1 1 0 sin(0 1 100 0 0)
rload 1 0 5000
.tran 0.5m 20m
.plot tran v(1)
.end
```



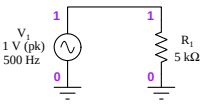


This is analogous to a prism refracting a beam of monochromatic light: the resulting spectrum consists of a narrow beam (peak) at one color (one frequency). In other words, this signal is a "pure" sinusoid at 100 Hz.

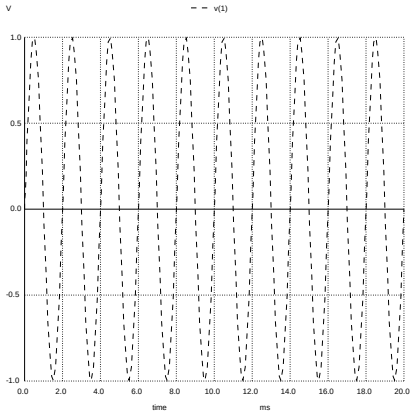


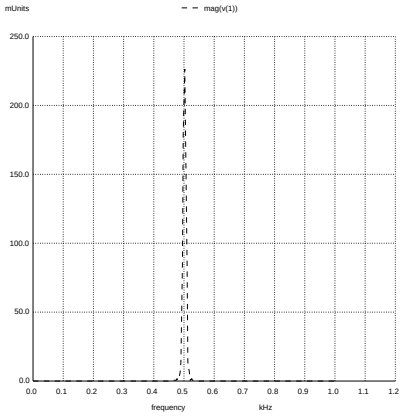
[2] In order to obtain crisp spectra, the transient analysis parameters of the NGSPICE netlists had to be modified from the values used to give each time-domain plot (oscillograph). Each spectrum was plotted using transient parameters of 0.5 millisecond intervals and an analysis period of 150 milliseconds (i.e. `.tran 0.5m 150m`). The sequence of NGSPICE commands used to generate these spectra (after reading netlist file with the `source` command and "running" the simulation with the `run` command) are as follows: `linearize v(1) ; fft v(1) ; plot mag(v(1))`.

Next we will explore the faster sine wave, defined in our simulation as having a peak amplitude of 1 Volt and a frequency of 500 Hz. Note how once again the spectrum consists of a single peak, this time shifted horizontally to align with the 0.5 kHz mark rather than the 0.1 kHz mark as before:

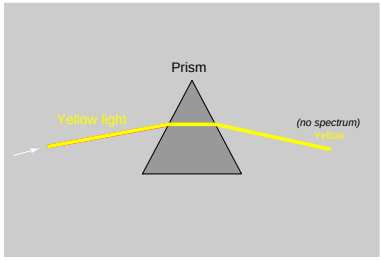



```
* Sine wave simulation
v1 1 0 sin(0 1 500 0 0)
rload 1 0 5000
.tran 0.1m 20m
.plot tran v(1)
.end
```

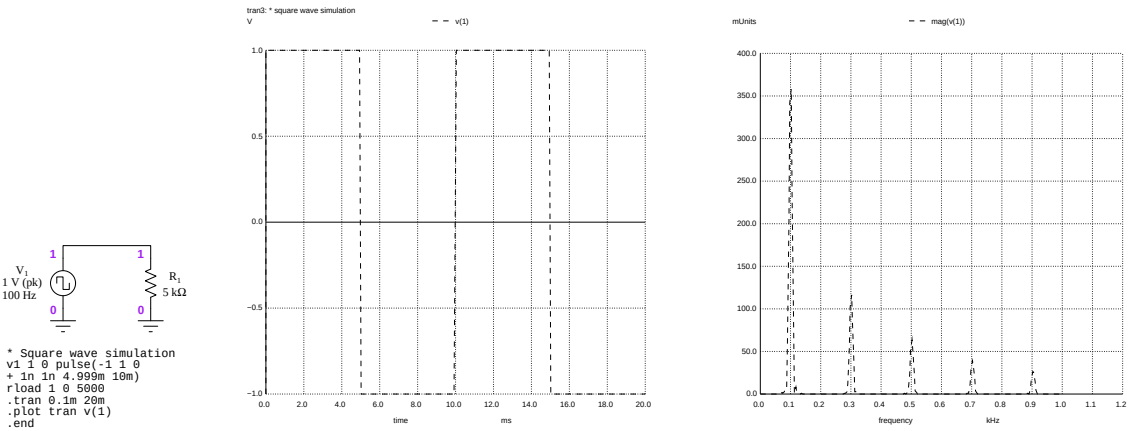




Once again the single spectrum peak is analogous to a prism refracting a beam of monochromatic light, having a color (frequency) different than before. Whereas the prism would refract this new monochromatic color by a new angle, the spectrum analyzer shows the peak located at a new place along the frequency-domain axis.



Our spectrum becomes more interesting, though, once the signal waveshape deviates from that of a sinusoid. Here, we see our simulated square wave (1 Volt peak, 100 Hz frequency) producing spectrum of several peaks rather than one peak:



Examining the oscillograph, we see how the period of the square wave is 10 milliseconds. This reveals its *fundamental* frequency to be 100 Hz ($f = \frac{1}{t}$). The spectrum analysis does show a peak at 100 Hz, but it also shows peaks at 300 Hz, 500 Hz, 700 Hz, and 900 Hz. These integer-multiples of 100 Hz are called *harmonics*, with the fundamental (100 Hz) being the first harmonic, 300 Hz being the 3rd harmonic, 500 Hz being the 5th harmonic, and so on. This result is analogous to a prism splitting up a light beam into several distinct beams, each one having its own angle and color (frequency), with the higher-frequency colors being weaker than the others.

An oscilloscope and a spectrum analyzer reveal different perspectives of the same thing. In this case, a square wave electrical signal *is* a voltage that switches from one extreme value to the other at regular intervals over time, but that same signal *also is* a superposition of sinusoidal voltage waves at different frequencies and amplitudes. It is imperative to understand that a spectrum analyzer does not alter the wave at all, any more than a prism modifies a beam of light: both instruments merely give us a "disassembled" view of what always existed inside the waveform. A waveform *is* an amplitude varying over time, and it *is* a series of sinusoidal waves superimposed upon one another.

While the mathematical principles linking time-domain and frequency-domain representations ($f(t)$ and $f(\omega)$, respectively) are quite complex[3], it is possible to articulate several rules describing the relationship between time- and frequency-domains that do not require advanced mathematics to apply to practical scenarios. Additionally, some of the practical methods developed to display a frequency-domain spectrum from a time-domain signal are actually easier to understand than the underlying mathematics, much the same as a glass prism is easier to apprehend and use than the mathematics of light waves!

[3] Jean Baptiste Joseph Fourier actually developed this mathematical technique for modeling the flow of heat through solid materials based on sinusoidal functions.

4.9 Time-frequency relationships

The following rules describe some of the relationships between time-domain and frequency-domain representation of waves, beginning with our previous statement that all waves are equivalent to combinations of sinusoids. No attempt will be made here to derive or to prove any of these rules in this tutorial, though they may all be confirmed by experiment:

- **Equivalence**: Any wave-shape, no matter how complicated, is equivalent to a sum of purely sinusoidal waveforms. The number of terms in this sum range from one to *infinite*.

  → A pure sinusoid with constant period has, by definition, just one frequency. It is a "sum" of exactly one sinusoidal waveform.

  → Any waveform that is not a perfect sinusoid contains multiple frequencies; i.e. it is exactly equivalent to a sum of perfect sinusoids.

- **Periodic waves**: Any *periodic* waveform (i.e. a waveform that repeats itself over time) is equivalent to a sum of cosine and sine waves, the frequency of each being a harmonic (i.e. an integer multiple) of the wave's fundamental frequency, and each with its own amplitude. This sum may also include a DC term.

- **Non-periodic waves**: Any *non-periodic* waveform is equivalent to a sum of sinusoids with frequencies not limited to just integer multiples of the harmonic, and may also contain a DC term. These non-integer frequencies are called *interharmonics*, and may exist as specific frequencies or as a continuous band covering a range of frequency.

- **Superposition**: Any superposition of waveforms, no matter how complicated, will result in a superposition of those waveforms' spectra: the whole is equal to the sum of its parts.

- **Linear systems**: Any signal passing through a *linear*[4] system will emerge with the same frequencies in its spectrum, although amplitudes and/or phase shifts may differ.

- **Non-linear systems**: Any signal passing through a *nonlinear* system will emerge containing new frequencies, and also a different wave-shape as viewed in the time domain.

- **Steepness**: Wave-shapes with shorter rise- and fall-times (i.e. steeper edges) in the time domain have more high-frequency components than wave-shapes with more gradual profiles, all other factors being equal.

  → *Limit case*: A pulse of infinitesimal width in the time domain contains *all* frequencies – its frequency-domain spectrum is a flat, horizontal line.

  → *Limit case*: An unchanging (DC) signal's frequency-domain spectrum is a single peak at 0 Hz.

- **Symmetry**: If a signal viewed in the time domain has a wave-shape that is symmetrical about its centerline – that is to say, if the shape of the wave is identical when inverted – then its spectrum will only contain *odd* harmonics.

[4] A "linear" system is one with a constant ratio of output to input. A resistor is a good example of a linear component, as its resistance R is a constant ratio between voltage V and current I: doubling the current through a resistor results in double the voltage drop. A semiconductor diode is a good example of a nonlinear component, where the voltage drop does not double in response to a doubling of current.

4.10 Applications for frequency-domain analysis

Now that we realize some of the relationships between time-domain and frequency-domain representations of waveforms, we should explore some practical applications:

Application – *Extracting frequency-domain information from time-domain measurements*: Computer algorithms exist to derive the frequency spectrum from any given signal sampled in its time domain. The most popular of these is the *Fast Fourier Transform* (abbreviated *FFT*), so named because it computes a spectrum with a remarkable economy of calculations. In fact, the spectrum plots shown earlier in this tutorial were created using an FFT algorithm in the SPICE circuit simulation software.

Application – *Linear circuit analysis by Superposition Theorem*: Recall the Superposition Theorem as first learned with DC circuits. The purpose of this Theorem was to simplify the analysis of a multi-source circuit by considering just one source at a time, knowing that for any linear network the results would be the superposition (i.e. algebraic sum) of the individual sources' effects. Now that we know any non-sinusoidal signal is a superposition of sinusoids, we may do the same: analyze any linear network *one harmonic at a time* and sum the results of all harmonics' effects on the circuit. This strategy allows us to use the same mathematical tools (e.g. phasors) learned for analyzing simple AC circuits.

Application – *Manually identifying signals within noise*: A signal waveform combined with extraneous waveforms (i.e. "noise" from our perspective) may be difficult if not impossible to visually discern from a time-domain oscillograph, but will be much easier to identify within a frequency-domain spectrum plot. Furthermore, the spectrum display helps us determine the noise frequencies, and therefore useful filtering strategies for mitigating that noise.

Application – *Manually identifying distortion*: Linear circuits are not supposed to distort the wave-shape of an AC signal passed through, so if there is any distortion present we would do well to know of it. Small amounts of distortion imposed on a "smooth" test wave-shape such as a sinusoid[5] may be difficult if not impossible to identify, much less *quantify* from a time-domain display. However, distortion is impossible to ignore in a frequency-domain display because it consists of *new frequencies* (i.e. peaks in the spectrum where none existed before), and the height of those new peaks reveals exactly how much distortion exists.

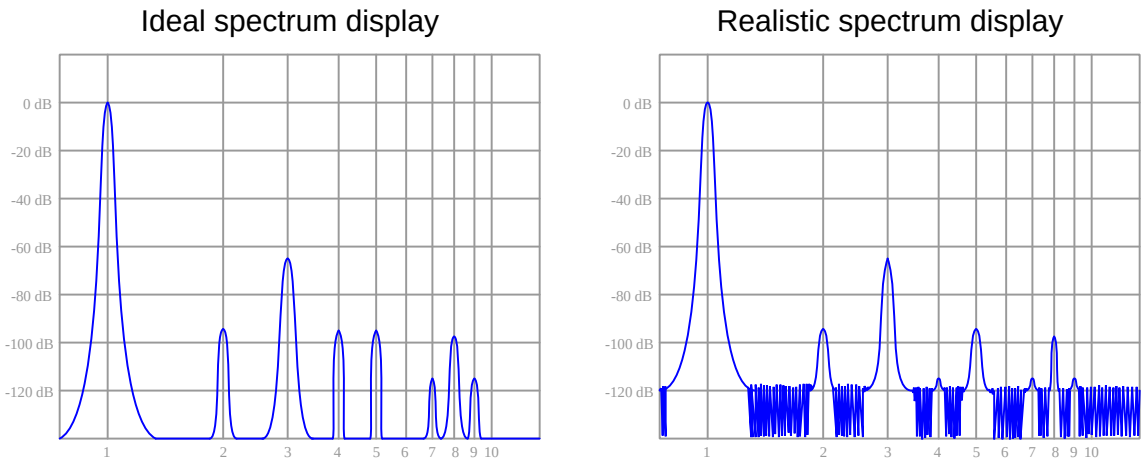
Application – *Diagnosing power-control circuits*: Electronic power-control circuits minimize resistive energy loss (dissipation) by switching their transistors and/or thyristors between extreme states of cutoff (fully off) and saturation (fully on). This results in current and voltage waveforms with fast rise- and fall-times, which we now know is equivalent to a rich spectrum of harmonic frequencies. The magnitudes and frequencies of these harmonics serve as indicators of switching behavior. For example, if a power control circuit operating on AC is designed to symmetrically "chop" the AC waveform, we would expect only odd-numbered harmonics. The presence of any

[5] This is why a common practice for identifying distortion in electronic circuits using only time-domain instruments is to use square-wave test signals rather than sinusoidal test signals. Simply put, it is easier to visually discern distortion in a square wave because the wave's features will no longer be square and plumb. This technique is limited, though, as some forms of distortion such as "crossover distortion" are much more challenging to discern than others due to the fast rise and fall times of square waves.

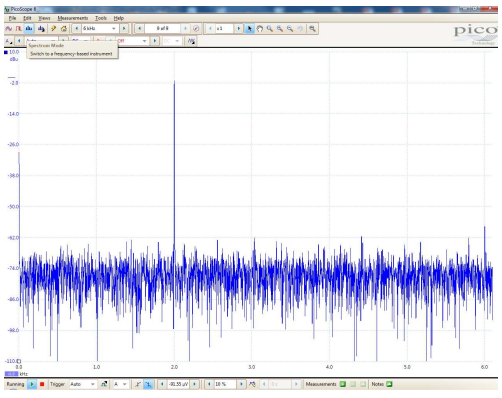
even-numbered harmonics in the spectrum indicate a lack of symmetry in the waveform, which could mean a problem in one-half of the power control circuit.

A practical parameter of real-world spectrum analysis is *random noise*. These are variations in the measured signal (voltage or current) lacking any distinct frequency, and as such it appears at the bottom of the spectrum display as a "jagged" or "fuzzy" baseline rather than the clean and straight lines one obtains from a computer-simulated spectrum plot.

The following illustrations show ideal (left) and real (right) displays of a spectrum showing fundamental and harmonic frequencies:



Any signals with peaks at or below the top of this "noise floor" of the display are swamped by the noise and cannot be measured. A screenshot of a spectrum analyzer measuring the output of a signal generator set to 2 kHz (sinusoidal wave) clearly shows the signal's peak amidst the noise floor:



When measuring a live signal, the individual negative peaks within this noise floor randomly come and go. It is not a static display, but an ever-shifting array of noise.

4.11 Filter-based Fourier analysis

As useful as it may be for us to know that any wave-shape is mathematically equivalent to a superposition of sinusoidal waves at different frequencies and amplitudes, it is even more useful to be able to analyze a given waveform to determine exactly what those sinusoidal frequencies and amplitudes are. This is called *Fourier analysis*, and its mathematical foundations are quite complex. The integral calculus necessary to even begin to approach this topic is well beyond the scope of this text, but fortunately there are analog electronic methods for determining the harmonic content of real AC electrical signals requiring no calculus to comprehend.

Perhaps the simplest method for analyzing an arbitrary AC signal is to construct a circuit with several band-pass filter networks, each one tuned to a different harmonic frequency of the known fundamental. This approach only works when the fundamental frequency is a fixed and known quantity, but for certain practical applications such as AC power systems where the fundamental frequency never varies (typically 50 Hz or 60 Hz depending on the geographic location) it is practical.

The following illustration comes from Charles Proteus Steinmetz's 1917 text *Theory and Calculation of Electric Circuits*, where he refers to each of the *LC* networks as a *wave screen*:

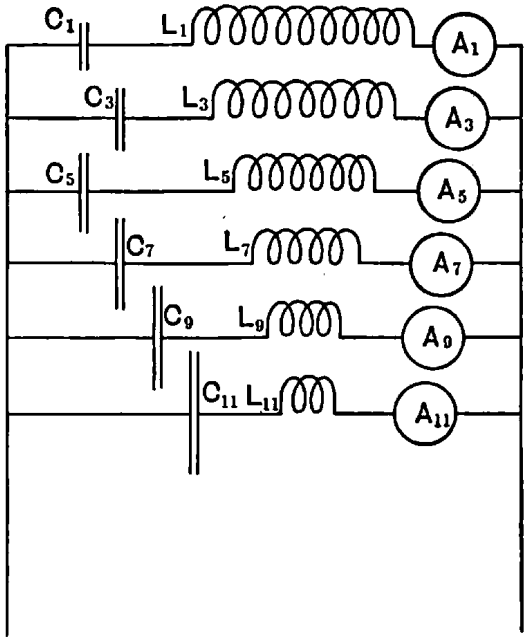


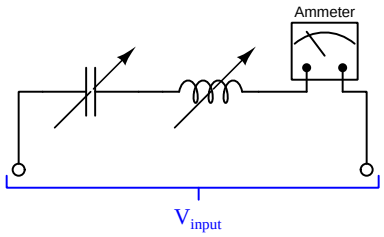
Fig. 76.

Each of the series *LC* networks is a band-pass filter allowing current through to a dedicated AC ammeter. Upon connecting this network's bottom two terminals to an appropriate source of power-line AC voltage, we would see the first ammeter (A_1) register the strongest value while the other ammeters register lesser amounts of current. That first ammeter's filter network (C_1 and L_1) of course is tuned to the power system's fundamental frequency, such that only the fundamental

component of the AC waveform powers ammeter A_1. The next filter network (C_3 and L_3) have values chosen to pass only the third harmonic of the system's fundamental frequency, so that ammeter A_3 measures only how much 3rd harmonic there is in the signal. All the other filter networks are similarly tuned to odd-numbered harmonics. One could expand this circuit to include even-numbered harmonic filters with more ammeters, but generally in power systems we find distortions to the AC generators' waveforms are symmetrical in the positive and negative half-cycles, and so the major harmonics of concern are all odd.

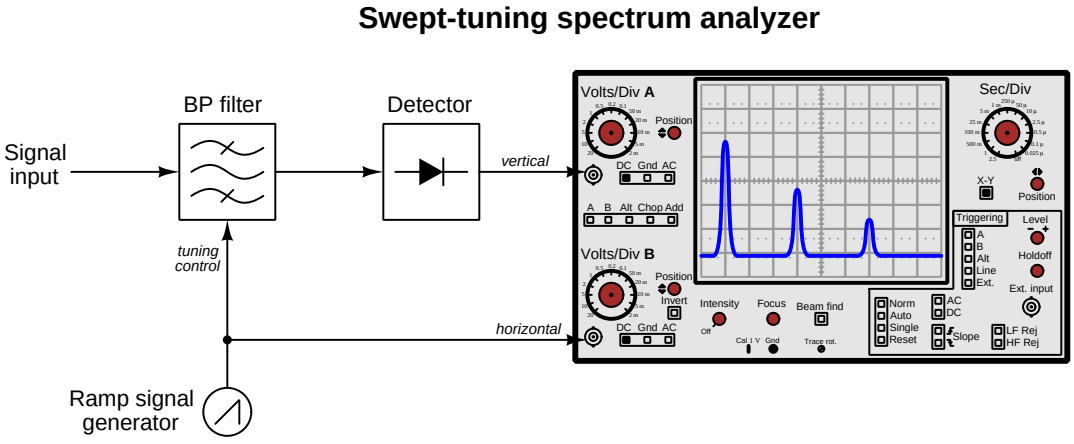
Steinmetz's "wave-screen" harmonic analyzer is simple and easy-to-understand but limited in that it only works for AC systems of a known and fixed fundamental frequency. We will need different analytical techniques in order to construct a more general harmonic analyzer.

One alternative using similar technology employs *variable* inductors and/or capacitors along with a single ammeter so that the band-pass filter may be manually tuned across a range of frequencies while carefully noting those frequencies where the ammeter's reading peaks:



This instrument has the advantage of being infinitely adjustable for analysis of harmonics covering a range of fundamental frequencies, the limits of that range bound by the variable L and C component adjustment ranges and by the ammeter's frequency range (i.e. *bandwidth*). However, it requires patience and skill to operate, unable to provide simultaneous measurements of harmonics the way the multi-filter analyzer can.

One improvement to this technique is to use an electrically-tuned[6] filter driven by a ramping voltage waveform, using that ramping signal to drive the horizontal axis of an oscilloscope (set to "X-Y" mode[7]) while the filtered signal amplitude is "detected" (i.e. rectified and peak-captured) and used to drive the oscilloscope's vertical axis. We see how such a *spectrum analyzer* system could be constructed in block-diagram form:



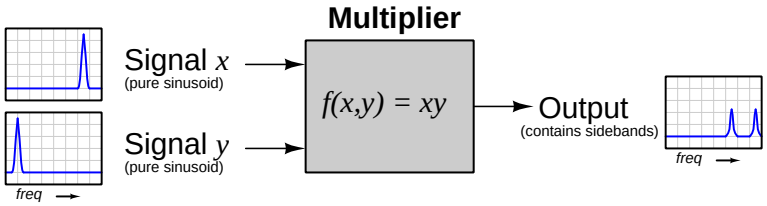
As the ramp generator's signal voltage increases over time, the filter's center frequency increases (selecting higher-frequency harmonics contained within the input) while the oscilloscope's trace sweeps from left to right. Whatever makes it through the band-pass filter at any given time is "detected" and converted into a positive DC voltage to drive the oscilloscope's trace upwards. The result is that a complete spectrum is repeatedly traced on the oscilloscope screen. Essentially, this swept-tuning analyzer does automatically what the manually-tuned band-pass filter did manually, with much faster results such that the peaks on the displayed spectrum appear to be simultaneous to any human viewer if the ramp signal's frequency is sufficiently high.

[6] Several techniques exist to make a filter network electrically tunable. This could be done mechanically, using a servo motor to turn the shaft of a variable inductor or capacitor. It could be accomplished through the use of a *varactor diode* whose capacitance changes with applied voltage.

[7] In this mode, the oscilloscope's horizontal (x) axis is driven by an external voltage rather than being swept by its own internal ramp signal, thus allowing the external ramp signal generator shown here to control the trace's horizontal position while the "detected" filter-output signal drives the vertical.

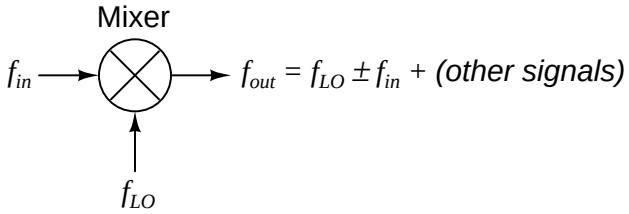
4.12 Mixer-based Fourier analysis

Another analog electronic method for analyzing waveforms' harmonic content utilizes *signal multiplication*. As it so happens, when we multiply two sinusoidal waveforms together in real time, the result is a pair of new sinusoidal waveforms: one having a frequency equal to the *sum* of the two multiplied-signal frequencies and another having a frequency equal to the *difference* between the two multiplied-signal frequencies. For example, if we multiplied a 10 kHz sinusoid by a 12 kHz sinusoid, the output would be a superposition of a 2 kHz sinusoid and a 22 kHz sinusoid:



These two sinusoids are known as *sidebands*, because they appear "off to either side" of the greater input frequency when viewed in the frequency domain (e.g. a lower sideband 10 kHz below the 12 kHz peak and an upper sideband 10 kHz above the 12 kHz peak). This phenomenon is generally known as *heterodyning*, and it occurs whenever signals "mix" together within a non-linear network. A purely multiplicative circuit is nonlinear, but so are many others, and any circuit using any form of non-linear characteristic to heterodyne two or more signals is generally known as a *mixer*.

Mixers are typically represented in block-diagram form by a circle with an "X" symbol inside, referencing the multiplicative characteristic necessary for heterodyning. Most mixer circuits are not *purely* multiplicative, which means their output signals usually include other frequencies in addition to the two sidebands of interest. The signal mixed with the input is commonly referred to as the *local oscillator*, or *LO*:

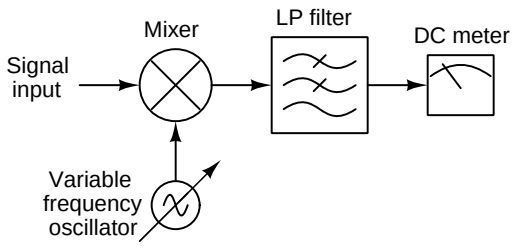


As we shall soon see, the presence or absence of these other frequencies are of no concern to us when we use a mixer for Fourier analysis.

Imagine passing two sinusoidal signals of precisely equal frequency into a mixer (i.e. $f_{in} = f_{LO}$). The two sideband frequencies output by that mixer will of course be the sum and difference frequencies, and in this case the result will be an upper sideband frequency of $2f$ ($f_{in} + f_{LO}$) and a lower sideband frequency of zero ($f_{in} - f_{LO}$). What a frequency value of "zero" means is that the lower sideband will be DC rather than AC. This is remarkable because it only occurs when two signals *of equal frequency* pass through a mixer. If the mixer's two signals differ at all in frequency there will be no DC present in the mixer's output because both sidebands will have non-zero frequency values.

If we monitor the output of a mixer using an instrument that only responds to DC signals, we may use that mixer to "probe" for the presence of particular frequencies within a complex signal. Consider for example a complex waveform having a fundamental frequency of 15 kHz along with all the even- and odd-numbered harmonics (e.g. 30 kHz, 45 kHz, 60 kHz, etc.), and passing this complex signal to the input of a mixer circuit. Then, imagine connecting a variable-frequency sinusoidal oscillator to the LO port of that same mixer circuit. Our DC-only sensing instrument will register when our variable oscillator reaches 15 kHz, and then again when the variable oscillator is dialed up to 30 kHz, then again at 45 kHz, etc. In other words, if we "sweep" the LO frequency across a range while monitoring the mixer's output DC, we will see DC signals only when the LO matches some harmonic present in the complex signal.

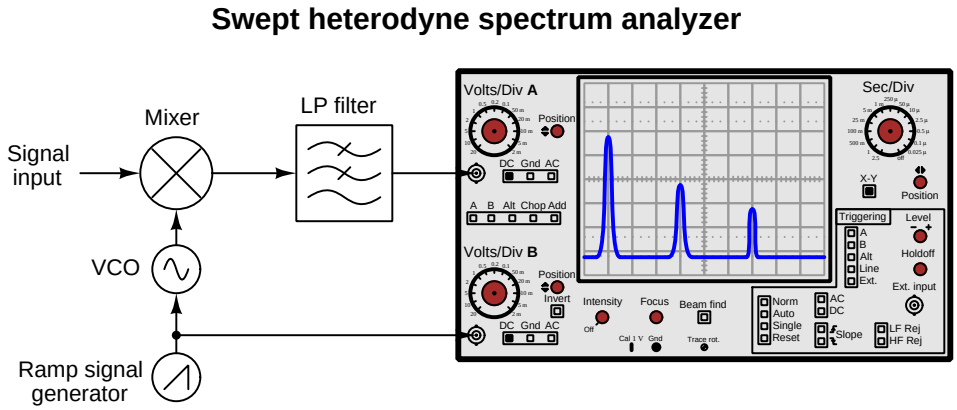
A block diagram of a manually-swept spectrum analyzer appears here, its operation similar to that of the manually-tuned LC filter and ammeter circuit shown in the previous section:



To operate this circuit, you would carefully monitor the DC meter's indication while gradually "sweeping" the variable oscillator's frequency, noting those frequency values where the DC meter gives a positive indication. Those will be the frequencies of every harmonic present in the input signal (i.e. the $n\omega$ values in the Fourier series equation), starting with the first harmonic or fundamental frequency. The magnitude of the meter's indication at each of these points represents the amplitude of each harmonic (i.e. the A coefficient values in the Fourier series):

$$A_0 + A_1 \sin(\omega t + \theta_1) + A_2 \sin(2\omega t + \theta_2) + A_3 \sin(3\omega t + \theta_3) + \cdots A_n \sin(n\omega t + \theta_n)$$

Just like the swept-tuning spectrum analyzer discussed earlier, we may similarly automate this mixer-based technique to display the resulting spectrum on an oscilloscope screen with that oscilloscope set to the "X-Y" mode as before. This time, in addition to sweeping the oscilloscope's trace from left to right, our ramp signal generator will also drive a *voltage-controlled oscillator* (*VCO*) which outputs an AC signal whose frequency value is proportional to that controlling voltage, which will then drive the mixer's local oscillator (LO) port. The low-pass filter's output will directly drive the oscilloscope's vertical input:



As the ramp signal generator's voltage ramps up from minimum to maximum, the VCO's output frequency will proportionately ramp from low frequency to high frequency as the oscilloscope's trace simultaneously moves from left to right on the screen. If the VCO frequency happens to match any harmonic frequency (including the fundamental) contained in the input signal, the mixer will output a DC voltage which passes through the low-pass filter to simultaneously drive the oscilloscope's trace upwards on the screen. However, if the VCO's frequency does not match any frequency contained in the input signal, then the mixer's output will be strictly AC (i.e. no DC) and the low-pass filter outputs little or nothing to the oscilloscope's y axis. Thus we end up with an image on the screen consisting of a peak at every horizontal point corresponding to a harmonic frequency contained in the input signal, and minimal height for the rest of the domain. If the ramp generator repeats itself at a fast enough rate, this spectrum display will appear continuously on the oscilloscope's screen as it re-traces the same shape over and over again.

4.13 Fourier analysis of a square wave

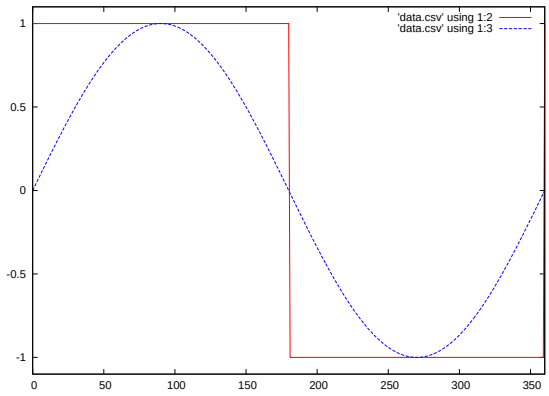
As we saw in the previous section, the mixer-based Fourier analysis technique involves multiplying the sample waveform by a pure sinusoid in real time (i.e. simultaneously), then analyzing the resulting product waveform for the presence of an average DC value to determine whether the frequency represented in that pure sinusoid is contained within the sample waveform. Now we will take this same strategy and manually apply it to the analysis of a square wave.

Square waves are very well-understood from a frequency-domain perspective, consisting of a fundamental component plus all odd harmonics at inversely-proportional amplitudes following *Fourier series* shown below, where ω is frequency in radians per second and t is time in seconds:

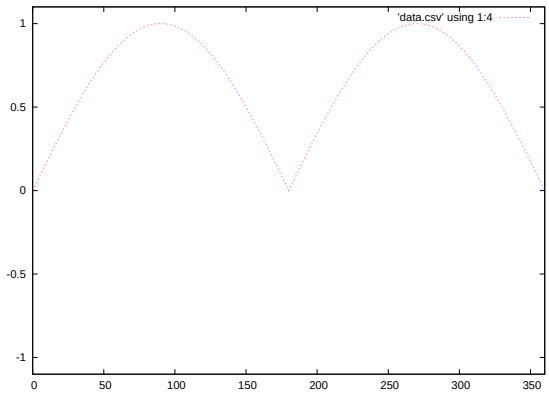
$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \frac{1}{7} \sin 7\omega t + \cdots + \frac{1}{n} \sin n\omega t$$

For example, a square wave having a fundamental frequency of 20 kHz will consist of a sine wave at 20 kHz added to a sine wave $\frac{1}{3}$ the amplitude at 60 kHz added to a sine wave $\frac{1}{5}$ the amplitude at 100 kHz, and so on to infinity. What we will now do is actually multiply a square wave by sine waves of different frequencies to prove to ourselves visually that the Fourier series as written for a square wave is true.

First we will consider multiplying a square wave by a sine wave having exactly the same (fundamental) frequency:

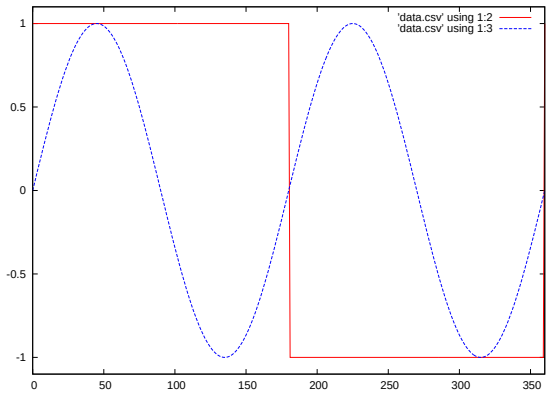


This square wave's value oscillates between +1 and −1, which means for the first half of its cycle the product of square times sine wave will be identical to the first half of the sine wave, but for the second half of the cycle the product will be the inverse (negative) of the sine wave. The result is what appears to be a fully-rectified sine wave, with both of its half-cycles positive rather than alternating positive and negative:

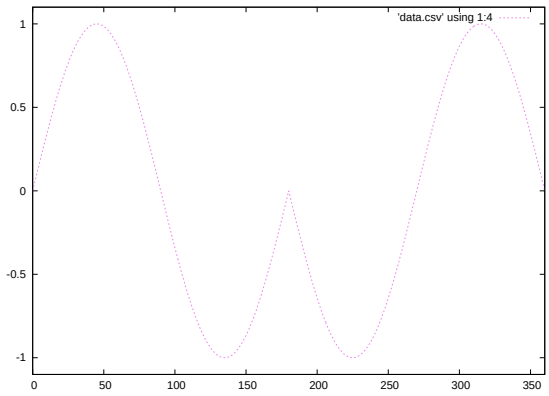


In order to determine whether or not an average DC value exists in this product, all we need to do is assess whether the average area encompassed by the wave (i.e. between the wave's curve and the horizontal centerline) has a net value or whether the positive areas completely cancel out the negative areas. It should be clearly evident in this case that a net positive value does exist because the product waveform never goes negative at all. The existence of an average DC value proves the square wave contains this particular sine wave's frequency.

Next we will increase the frequency of the sine wave to be twice that of the square wave's fundamental. This will be our test for a second harmonic within the square wave:

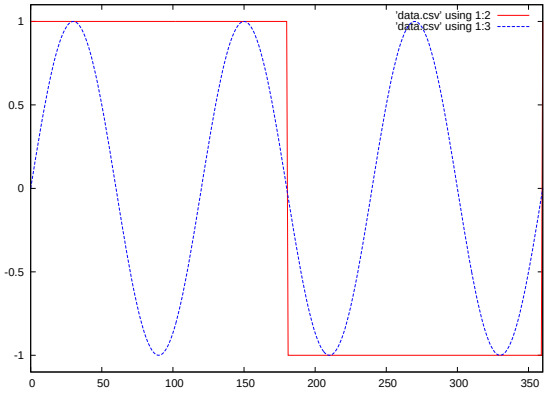


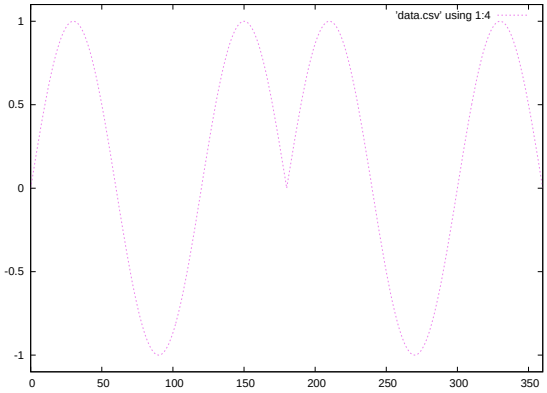
As before, when the square wave is in its +1 portion, the product will be an exact replica of the sine wave. When the square wave drops to a value of −1, the product will be an inverted version of the sine wave:



Looking carefully at this product waveform reveals four peaks of equal area, two positive and two negative. This means there is no average DC value to the product waveform, and therefore the sine wave signal used to generate it is *not* present within the square wave signal. Simply stated, a square wave contains no second harmonic.

Next we will increase the frequency of the sine wave to be three times that of the square wave's fundamental. This will be our test for a third harmonic within the square wave, the input signals shown first and the product shown next:



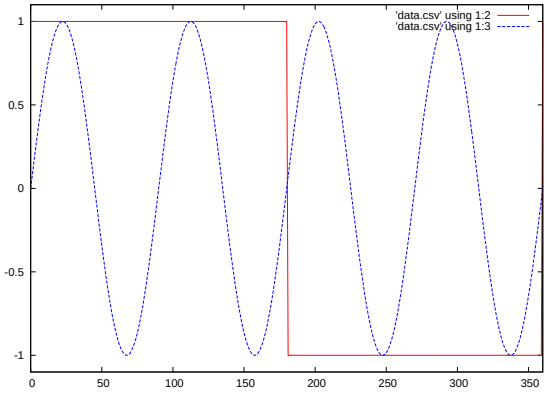


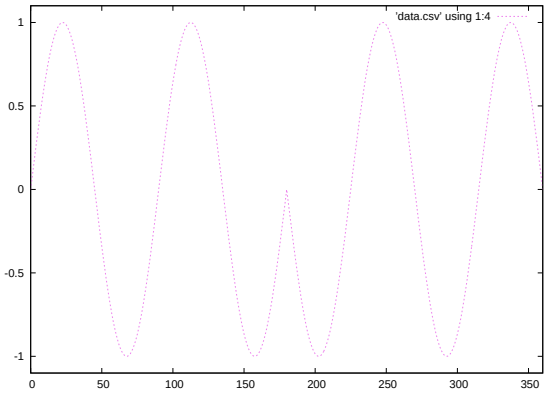
This time we see six peaks of equal area, four positive and two negative. The two negative peaks' areas cancel out two of the four positive peaks' areas, leaving a net positive value for the product waveform's enclosed area (i.e. it contains DC), proving a third harmonic exists within the square wave. Moreover, we have enough information here to quantify how strong this third harmonic is compared to the fundamental. Recall that in the first harmonic (fundamental) test we obtained a product waveform consisting solely of two positive peaks. In this test we also have a net area equal to two positive peaks, but each of these peaks is only one-third as wide (but equal height) as each peak seen in the first-harmonic test. This means the average DC value of this product waveform is just one-third as much as that of the product waveform created for the fundamental test.

What we have proven so far are the first two terms of the Fourier series for a square wave, namely that the third harmonic has just one-third the amplitude of the first harmonic (fundamental):

$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \cdots$$

Next we will increase the frequency of the sine wave to be four times that of the square wave's fundamental. This will be our test for a fourth harmonic within the square wave, the input signals shown first and the product shown next:



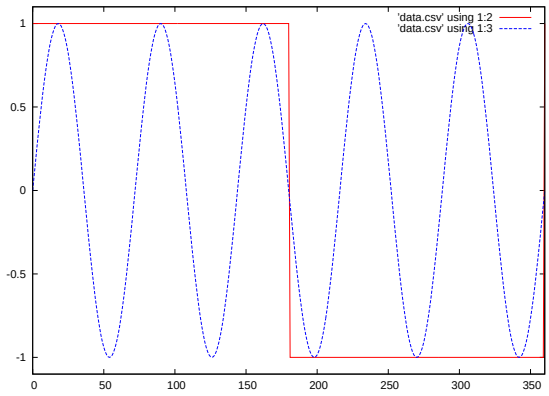


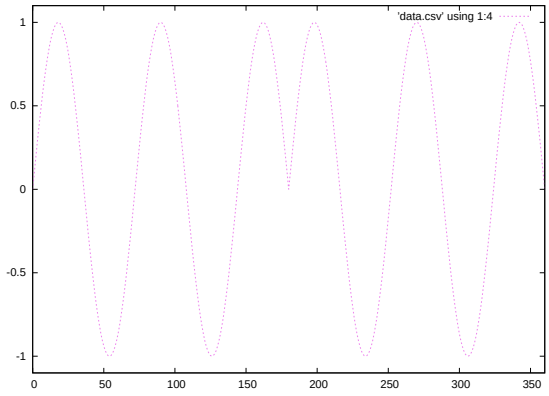
As with the second harmonic test, we see an equal number of positive and negative peaks. This means zero average DC value, and therefore the square wave does not contain a fourth harmonic.

If we think carefully about the results so far, it should be evident that *no* even-numbered harmonic will exist within the square wave because within each half-cycle of the square wave there will always fit a whole number of even-harmonic sine wave cycles. The areas enclosed by the product waveform's peaks during the first half-cycle of the square wave will all cancel because every positive peak is matched by an identical negative peak, and the same will be true during the square wave's second half-cycle as well. The fact that the sine wave gets inverted during the second half-cycle is of no consequence to the net area value, so long as a whole number of sine wave cycles fit within each half-cycle of the square wave.

This means it is pointless to test for a sixth, eighth, tenth, or further even-numbered harmonic. All their net areas in the product waveform are guaranteed to be zero, meaning the square wave does not contain any even-numbered harmonics.

Next we will increase the frequency of the sine wave to be five times that of the square wave's fundamental. This will be our test for a fifth harmonic within the square wave, the input signals shown first and the product shown next:





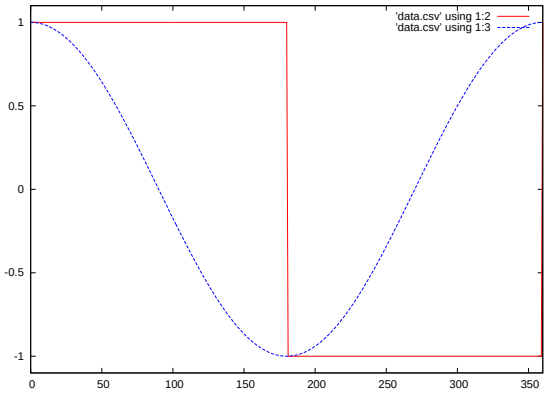
Here we see a net positive area enclosed by the product waveform: six positive peaks and four negative peaks. This leaves a net of two positive peaks, each of those peaks enclosing an area just one-fifth of the peaks seen in the product waveform for the first-harmonic (fundamental) test. Therefore, the square wave's fifth harmonic has an amplitude just one-fifth as much as its fundamental, completing another term in the Fourier series:

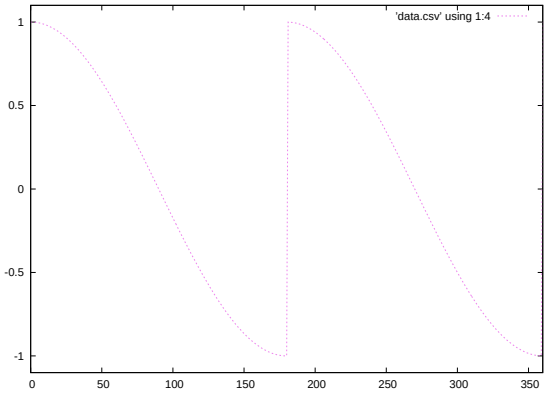
$$\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \cdots$$

We could continue this process for all the other odd-numbered harmonics, but this is not necessary once you recognize the pattern. With each subsequent odd-numbered harmonic test, we find an average DC value within the product diminishing in inverse proportion to its harmonic number, and this series extends to infinity because there must *always* be some amount of DC after multiplication by any odd-numbered harmonic.

4.14 Phase-shifted Fourier terms

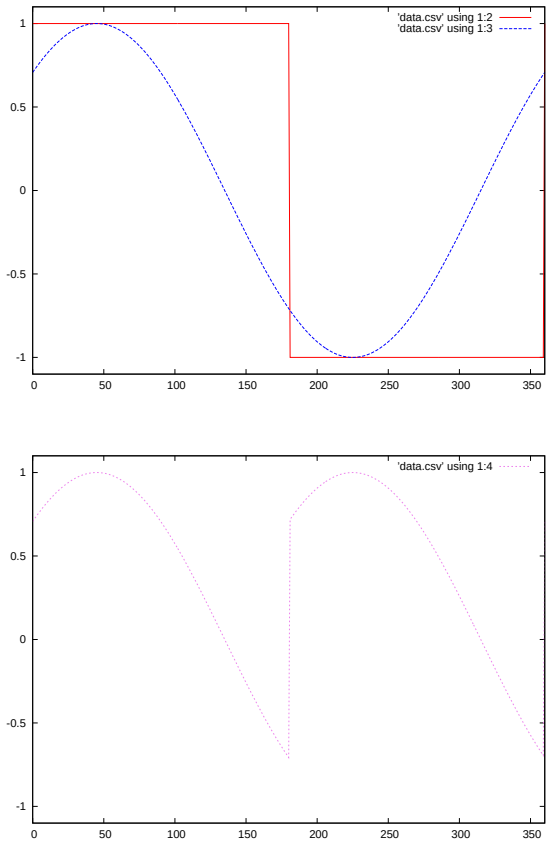
The examples we just saw for the 1st, 2nd, 3rd, 4th, and 5th harmonic tests of a square wave should make a compelling case for how Fourier analysis works for any real signal, but there is an important caveat regarding the *phase relationship* between the mixer's sampled waveform (input signal) and the sinusoid (Local Oscillator). Let's explore this by calculating the product of the same square wave and a *cosine* wave of identical frequency, first the input signals and then the product waveform:





Here we see equal positive and negative areas for the product waveform, which means no DC average value and therefore no presence of this harmonic within the square wave. At first this may sound like a contradiction, because we clearly saw a strong net positive area in the product waveform when we used a sine wave instead of a cosine wave at the fundamental frequency, proving that a first harmonic does exist within a square wave. The fact that we find no DC average now proves that the phase-shift of the sinusoidal test signal (i.e. the Local Oscillator, or LO, signal) in relation to the sampled waveform actually matters. As such, *we may miss the presence of a harmonic* in the sampled waveform if we happen to choose a LO test signal having the wrong phase shift.

Exploring this phase-shift concept a little more, we will re-run the signal multiplication using a test signal phase-shifted by 45^o instead of 0^o (sine) or 90^o (cosine):



Here, the product waveform definitely has a greater positive area than negative area, which means a net average DC value exists, albeit not as strong as when we used a plain sine wave as the LO test signal.

If we think about this carefully, we can generalize that a plain sine wave multiplied by a square wave must give us the strongest positive net area (and therefore the strongest average DC value) for the product waveform, but that this area diminishes as the phase shift approaches 90^o. We may even propose using an inverted sine wave as the LO test signal (i.e. 180^o phase shift), in which case the product waveform would have an entirely *negative* net area. The average DC value we get out of the multiplication function varies from full positive to full negative if that harmonic exists in the sample waveform depending on the phase angle of the LO signal we happen to choose for the test.

What is happening here is that the harmonic itself we are trying to find within the sampled waveform has a definite phase angle in relation to that sampled waveform, and that we obtain a maximum positive DC value from the product waveform only when our LO test signal matches the phase of that harmonic. We cannot simply assume every harmonic will reveal itself by using sine waves for the LO signal in every test.

This is why the generic Fourier series contains a phase-shift value (θ) for each harmonic as well as an amplitude (A) and a frequency value ($n\omega$), because sometimes harmonics require a certain amount of phase shift to properly synthesize the whole waveform:

$$A_0 + A_1 \sin(\omega t + \theta_1) + A_2 \sin(2\omega t + \theta_2) + A_3 \sin(3\omega t + \theta_3) + \cdots A_n \sin(n\omega t + \theta_n)$$

Recall that a more verbose form of the generic Fourier series shows each harmonic as being the sum of a cosine term and a sine term, each with a common frequency value but having independent amplitudes (a and b respectively):

$$a_0 + (a_1 \cos \omega t + b_1 \sin \omega t) + (a_2 \cos 2\omega t + b_2 \sin 2\omega t) + (a_3 \cos 3\omega t + b_3 \sin 3\omega t) + \cdots (a_n \cos n\omega t + b_n \sin n\omega t)$$

We have already seen the mathematical basis for this equivalence in the form of complex numbers used in the *phasor* representation of AC quantities: we may represent a sinusoid at any given frequency in terms of its amplitude and phase angle ($A \angle \theta$) as a "polar" quantity, or in terms of its real and imaginary values ($a + jb$) as a "rectangular" quantity. The phase-shift version of the Fourier series is the "polar" representation of each harmonic, while the cosine-sine version of the Fourier series is the "rectangular" representation of each harmonic. Both forms are perfectly valid.

When analyzing a sampled waveform using the cut-and-try method of multiplying that waveform's values by corresponding values of a sinusoid as we did for the square wave, it would not be practical for us to try finding each harmonic's maximum product by trying every conceivable phase shift value until we determine which phase angle gives us the greatest average DC value in the product. However, performing a two-step test where we multiply by the cosine at that harmonic frequency, then again multiplying by the sine at that harmonic frequency, is manageable while providing us with all the information we need to determine the Fourier series. In other words, it makes more sense for us to analyze a sample waveform using the "rectangular" approach of the cosine-sine Fourier series form than the "polar" approach of the phase-angle Fourier series form.

4.15 A digital Fourier transform algorithm

Modern electronic test equipment is largely digital in nature rather than analog, but we may apply the same fundamental principle of mixer-based Fourier analysis – namely, multiplying a sampled waveform by "test" sinusoids of different frequencies and looking for an average DC value in the product – in the form of a digital computer program with similar results. In this section we will experiment with such an algorithm coded in the C programming language.

We will assume our algorithm obtains its waveform data from sampling an actual measured signal over time, those sampled values stored in a data structure known as an *array*. An "array" in the C language is a set of variables indexed by a subscript value, resembling subscripts used in mathematical formulae to distinguish related variables from each other. For example, if we refer to our sampled signal as x, then its instantaneous value at time zero would be x_0, its value at sample one would be x_1, its value at sample two would be x_2, and so on. In C, we would declare an array named `x` and then reference each of its elements as `x[0]`, `x[1]`, `x[2]`, etc.

Next, our algorithm must examine the values stored in this array, and multiply each of them by the values of a cosine function and a sine function at angles corresponding to those sample times. If one cycle of the sampled waveform requires 100 samples, then both the cosine and sine functions must complete one cycle in exactly 100 increments as well. An easy way to determine if there is an average DC value to these products is to compute a running total over all samples: if all the positive values cancel out all the negative values, the final sum will be zero; otherwise there will be some un-canceled value representing the amount of DC present in the product.

The following C program does all of this and more. In this example, we have encoded a square wave function as one hundred sample values all stored in the array `x[]` as double-precision floating-point quantities:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  double x[100] = {0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   0.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0};
  int sample;
  double cosavg, sinavg;
  cosavg = sinavg = 0.0;

  for (sample = 0 ; sample < 100 ; ++sample)
  {
    cosavg = cosavg + (x[sample] * cos(2 * M_PI * sample / 100));
    sinavg = sinavg + (x[sample] * sin(2 * M_PI * sample / 100));
  }

  printf("DC average value of cosine product = %f\n", cosavg / 100);
  printf("DC average value of sine product = %f\n", sinavg / 100);

  return 0;
}
```

The cosine and sine products and their running totals are computed within the `for()` loop, and then afterward those totals are divided by the number of samples (100) to obtain an average value for each. Note how the angular argument for the `cos()` and `sin()` functions are scaled such that 100 samples is equal to 2π radians, so that both trigonometric functions experience exactly one cycle over the total run of samples.

If we compile and run this code, we find it gives us a cosine average of zero and a sine average of 0.636410. Since the cosine and sine functions cover one cycle in the span of 100 samples, this means we are testing for the first harmonic (fundamental) within the sampled square wave.

A minor modification to this code allows us to re-analyze the sampled square wave for different harmonics. Here, we edit the `cosavg` and `sinavg` lines within the `for()` loop to detect a *third* harmonic:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  double x[100] = {0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   0.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0};
  int sample;
  double cosavg, sinavg;
  cosavg = sinavg = 0.0;

  for (sample = 0 ; sample < 100 ; ++sample)
  {
    cosavg = cosavg + (x[sample] * cos(3 * 2 * M_PI * sample / 100));
    sinavg = sinavg + (x[sample] * sin(3 * 2 * M_PI * sample / 100));
  }

  printf("DC average value of cosine product = %f\n", cosavg / 100);
  printf("DC average value of sine product = %f\n", sinavg / 100);

  return 0;
}
```

Now the `cos()` and `sin()` functions both cover three cycles over the 100-sample period of the square wave, thanks to the `3` multiplier inserted in those two lines of code.

When we compile and run this code, the results indicate a cosine product average of zero and a sine product average of 0.211578. This stands to reason, as the ratio $\frac{0.211578}{0.636410}$ is very nearly equal to $\frac{1}{3}$. Recall that the third harmonic of a perfect square wave has an amplitude exactly one-third that of the fundamental.

Testing for other harmonics is as simple as editing the leading coefficient inside the parentheses of the `cos()` and `sin()` functions. Testing an entirely different waveform is as simple as substituting

different values into the `x` array. Many other additions to this simple program are possible, a few of them listed here:

- Add code to re-test the sampled waveform for all harmonics within a certain range (instead of just one at a time).
- Add code to take the resulting `cosavg` and `sinavg` values and use them to compute a phasor magnitude and angle so we could write a Fourier series in "polar" form.
- Add code to take the computed harmonic amplitudes and normalize them to the first harmonic so that the fundamental always has an amplitude of 1, with all the other harmonics' amplitude coefficients being ratios of the fundamental's.
- Instead of having the number of signal samples (and the number of `for()` loop iterations, and the averaging divisor) fixed at 100, make that factor its own constant in the program. This will make it simpler to analyze waveforms recorded over different numbers of samples: just edit that constant value rather than having to make six different edits in the existing code.

Chapter 5

Historical References

This chapter is where you will find references to historical texts and technologies related to the module's topic.

Readers may wonder why historical references might be included in any modern lesson on a subject. Why dwell on old ideas and obsolete technologies? One answer to this question is that the initial discoveries and early applications of scientific principles typically present those principles in forms that are unusually easy to grasp. Anyone who first discovers a new principle must necessarily do so from a perspective of ignorance (i.e. if you truly *discover* something yourself, it means you must have come to that discovery with no prior knowledge of it and no hints from others knowledgeable in it), and in so doing the discoverer lacks any hindsight or advantage that might have otherwise come from a more advanced perspective. Thus, discoverers are forced to think and express themselves in less-advanced terms, and this often makes their explanations more readily accessible to others who, like the discoverer, comes to this idea with no prior knowledge. Furthermore, early discoverers often faced the daunting challenge of explaining their new and complex ideas to a naturally skeptical scientific community, and this pressure incentivized clear and compelling communication. As James Clerk Maxwell eloquently stated in the Preface to his book *A Treatise on Electricity and Magnetism* written in 1873,

> It is of great advantage to the student of any subject to read the original memoirs on that subject, for science is always most completely assimilated when it is in its nascent state . . . `[page xi]`

Furthermore, grasping the historical context of technological discoveries is important for understanding how science intersects with culture and civilization, which is ever important because new discoveries and new applications of existing discoveries will always continue to impact our lives. One will often find themselves impressed by the ingenuity of previous generations, and by the high degree of refinement to which now-obsolete technologies were once raised. There is much to learn and much inspiration to be drawn from the technological past, and to the inquisitive mind these historical references are treasures waiting to be (re)-discovered.

5.1 Wave screens

Charles Proteus Steinmetz was an electrical engineer employed for many years by the General Electric Company in New York. He was widely recognized as a genius in this field, and did much to elevate the mathematical rigor of electrical engineering. In his book *Theory and Calculation of Electric Circuits* first published in 1917 he describes the use of capacitance and inductance to form filtering circuits which he referred to as *wave screens* useful for separating alternating and direct current components of any pulsating electrical signal:

> **78.** By "wave screens" the separation of pulsating currents into their alternating and their continuous component, or the separation of complex alternating currents – and thus voltages – into their constituent harmonics can be accomplished, and inversely, the combination of alternating and continuous currents or voltages into resultant complex alternating or pulsating currents.
>
> The simplest arrangement of such a wave screen for separating, or combining, alternating and continuous currents into pulsating ones, is the combination, in shunt with each other, of a capacity, C, and an inductance, L, as shown in Fig. 75. If, then, a pulsating voltage, e, is impressed upon the system, the pulsating current, i, produced by it divides, as the continuous component can not pass through the condenser, C, and the alternating component is barred by the inductance, L, the more completely, the higher this inductance. Thus the current, i_1, in the apparatus, A, is a true alternating current, while the current, i_0, in the apparatus, C, is a slightly pulsating direct current. [page 156]

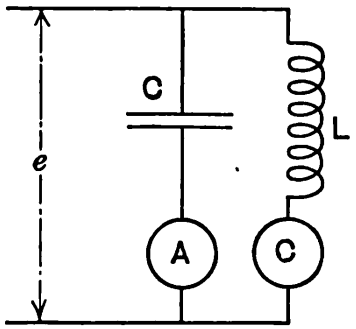


FIG. 75.

In this illustration A and C each represent electrical ammeters registering current through their respective branches of the parallel ("shunt") network.

On the next page, Steinmetz describes the use of series LC resonance to form band-pass filters useful for separating various harmonic[1] frequencies from a complex AC signal:

> Wave screens based on resonance for a definite frequency by series connection of capacity and inductance, can be used to separate the current of this frequency from a complex current or voltage wave, such as those given in Figs. 56 to 63, and thus can be used for the separation of complex waves into their components, by "harmonic analysis."
>
> Thus in Fig. 76, if the successive capacities and inductances are chosen such that
>
> $$2\pi f L_1 = \frac{1}{2\pi f C_1} \ ,$$
>
> $$6\pi f L_3 = \frac{1}{6\pi f C_3} \ ,$$
>
> $$10\pi f L_5 = \frac{1}{10\pi f C_5} \ ,$$
>
> $$2n\pi f L_n = \frac{1}{2\pi f n C_n}$$
>
> where f = frequency of the fundamental wave. [page 180]

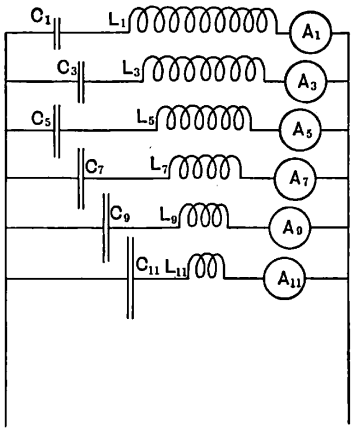


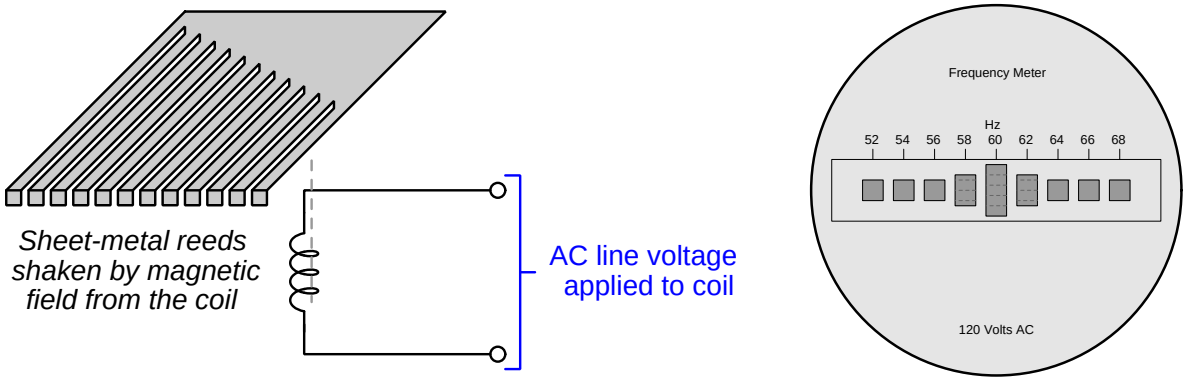
Fig. 76.

Steinmetz's conception of multiple band-pass filter networks tuned to resonate with respective harmonics of a known fundamental frequency, each one connected to its own ammeter to register the strength of each harmonic, is analogous to obsolete vibrating-reed frequency meters with their multiple reeds tuned to different frequencies.

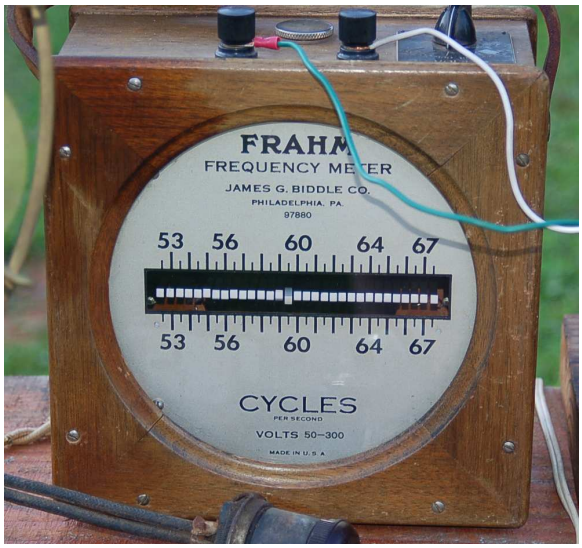
[1] As mathematically proven by Fourier, any periodic wave of any shape whatsoever is mathematically equivalent to the sum of a set of sinusoidal waves having frequency values equal to whole-numbered multiples of the fundamental frequency of the complex wave. For example, a complex-shaped waveform having a frequency of 45 Hz may consist of a 45 Hz "fundamental" sinusoid (the first harmonic) plus other sinusoidal waves of specific amplitudes having frequencies of 90 Hz (the second harmonic), 135 Hz (the third harmonic), 180 Hz (the fourth harmonic), etc.

5.2 Vibrating-reed meters as spectrum analyzers

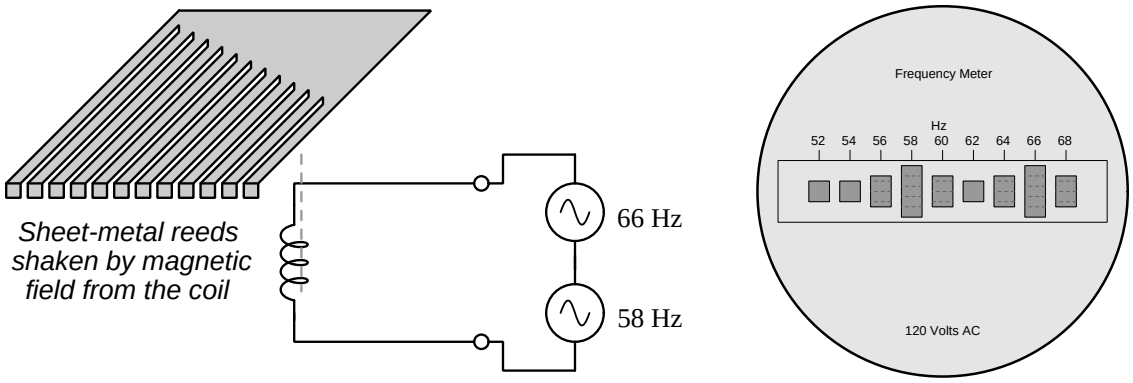
An obsolete technology for measuring the frequency of AC electric power is the *vibrating-reed* frequency meter. This meter consisted of a set of metal reeds made of spring-steel, each reed cut to a different length, and all of them excited by the magnetic field from an electromagnet coil energized by the AC line voltage to be measured. The illustration on the left shows the internal construction of the meter with its metal reed array and coil, while the illustration on the right shows what the meter looks like when energized by 60 Hz AC:



If the AC line frequency is 60 Hz, the reed tuned to resonate at that frequency will vibrate at the greatest amplitude, making the end of that reed appear "taller" as it rapidly shakes up and down. All the other reeds vibrate as well, but none as strongly as the reed resonating with the line frequency. A photograph of a real vibrating-reed frequency meter appears here, connected to an AC generator outputting approximately 59.5 Hz:



As crude a measuring instrument as a vibrating-reed frequency meter is, it actually functions as a sort of spectrum analyzer. A pure sinusoidal AC voltage has but a single frequency in its spectrum, and the moving reed ends reveal the outline of this spectrum: a single peak at the line frequency. If we were to connect a pair of AC generators in series and run them at different speeds to form a superposition of two sinusoidal voltages, we would see *two* reeds vibrate strongly on the face of such a frequency meter: two peaks in the spectrum, one for each generator's output:



For readers familiar with *filter* networks, a good way to model each of the metal reeds in such a meter is as a *mechanical band-pass filter*. Each reed has its own resonant frequency dictated by its mass, length, and elasticity. Adjusting the length of each reed is the simplest way to "tune" each of the reeds to the desired resonant frequency, which is why each reed in a meter such as this has a different length.

5.3 Wireless versus cable-based telephony

John Ambrose Fleming was a scientific researcher and prolific writer on electricity and early electronics, including communication systems. In his 1910 book *The Principles of Electric Wave Telegraphy and Telephony* Fleming discusses the limitations of communicating speech signals over long distances via wire cable, compared to transmitting them through space[2]. Note the reference to Fourier's equivalence between the waveform of an "articulate sound" and that of pure sinusoids, and how this perspective is used to explain the distorting effects of long wire cables.

> Although, therefore, wireless telephony has not attained the range reached by wireless telegraphy, nevertheless there is evidence that it has been conducted over distances of 200 miles or so with considerable success. One feature of importance in connection with radiotelephony is that there is no distortion of the wave form with distance. In ordinary telephony with wires, this distortion, which is specially marked in the case of circuits having large capacity, such as submarine cables, and the distorting power of the circuit impose a very serious limit upon the range of telephony. Briefly speaking, the reason for this is as follows:
>
> If we have a conductor with resistance R per unit of length, capacity C per unit of length, inductance L per unit of length, and dielectric conductance K per unit of length, then from the fundamental equations for the propagation of electrical disturbance along such a circuit (see Chap. IV., 1), it can be shown that the velocity of propagation of a periodic electric disturbance along the cable is a function of the frequency, the greater the frequency the less being the velocity of propagation. When articulate speech is being made against a telephone diaphragm, we have already seen that the wave form representing that articulate sound is a complex single valued curve, which in accordance with the theorem of Fourier can be analyzed into the sum of a number of simple constituent sine curves of different amplitudes differing in phase. These different harmonic constituents are propagated through the cable with different velocities and attenuated at different rates, so that when they are synthesized at the other end by the receiving telephone and the ear, the wave form which is reproduced is not an exact copy of the sound originally transmitted. In other words, we may say that the received sound is [page 863]
>
> a caricature of the sound made in the transmitting microphone. If the distortion is not too great, the ear at the receiving telephone is able to reconstruct or guess the sound at the other end creating it, just as we can recognize the original in a caricature of the human face if the caricaturing is confined within certain limits. Or to use another simile, in ordinary handwriting probably no single letter is quite correctly formed. If, however, the departure from perfect correctness is not too great, practice enables us to guess the word which is signified. The ear is therefore in this way able to recognize as a certain articulate sound, a sound which is not precisely like it. If, however, the distortion of the

[2] In Fleming's time it was widely believed that empty space was not empty at all, but in fact was filled with a mysterious substance called *aether*. The reasoning was that since electromagnetic radiation possessed wave-like characteristics, these waves required a substance to travel through just as waves in the ocean require water as their medium. The aether theory of electromagnetic waves was later disproved, but for the context of this discussion its accuracy is irrelevant.

wave form has proceeded beyond certain limits, the ear is no longer able to recognize the origin of the sound heard. Hence a serious limit is imposed upon telephony through the ordinary submarine cable.

It can be shown from the theory of such a cable, that if the relation between the four constant quantities R, L, C, and K is such that if $\frac{R}{L} = \frac{K}{C}$ then the cable will possess no distortion. In most ordinary cables the inductance is too small to fulfil this relation, but by the addition of inductance coils inserted at certain regular intervals in the cable, Pupin has been able to improve considerably telephonic speech through cables and long aerial lines. Such a cable is called a *loaded* cable. In the case, however, of wave transmission through the aether there is no distortion, because electromagnetic waves of all wave lengths travel through the aether at precisely the same speed, namely with the velocity of light. Accordingly, whatever may be the wave form of the wave which leaves the transmitting antenna, it will always preserve that same wave form although it may be attenuated or weakened by the diffusion of the energy over a large area. [page 864]

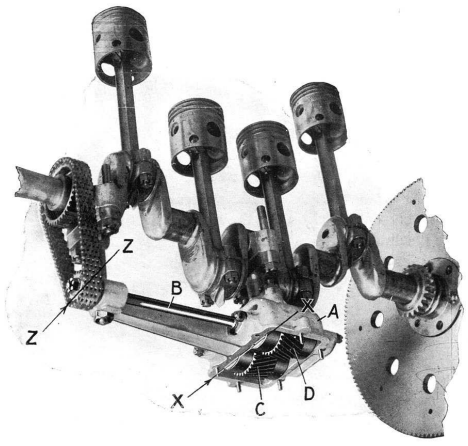
5.4 Reciprocating engine balance shafts

Reciprocating (i.e. piston-based) internal combustion engines naturally vibrate. With so many metal components moving to and fro, it is nearly impossible to build such an engine that runs perfectly smooth. This is especially true of inline-four-cylinder engines.

A simple technique for mitigating vibration in any rotating machine is to intentionally place *counterweights* on one or more of its rotating shafts, such that the vibration produced by this off-center weight will cancel the machine's natural vibration. This technique is limited, though, as a counterweight placed on a rotating shaft generates a purely sinusoidal vibration – that is, a vibration at the fundamental frequency of that shaft's rotation – and therefore may only cancel out the portion of the machine's vibration spectrum matching that particular frequency. An engine producing a complicated spectrum of vibrational frequencies cannot be "counterbalanced" with a single counterweight of any size in any location.

As mentioned, four-cylinder-inline engines are well-known for their strong vibration spectra, particularly for possessing a strong *second harmonic* vibration to the main crankshaft's rotation. The only way to counterbalance this particular frequency is to place a counterweight on a shaft spinning exactly twice as fast as the engine's crankshaft. Unfortunately, there is no normal component in an internal combustion engine designed to spin twice as fast as the crankshaft. An engine's *camshaft* spins at half the speed of the crankshaft, but that would do no good here.

Some four-cylinder-inline engines, therefore, are built with special second harmonic counterbalance units. A photograph of the rotating crankshaft and counterbalance assembly from an antique four-cylinder engine appears here, courtesy of H. Thornton Rutter's book *Modern Motors – Their Construction, Management and Control* Volume I published in 1922:



This engine's crankshaft is shown with pistons and connecting rods attached and aligned vertically (up from the crankshaft). No engine block hides our view of this assembly. The counterbalance drive shaft is labeled B, and spins at twice the crankshaft's speed by virtue of its small drive sprocket Z. Two counterweights (X) reside on gears C and D driven by shaft B. These two counterweights rotate in opposite directions in order to control the plane of their vibration: strictly vertical and not horizontal.

Chapter 6

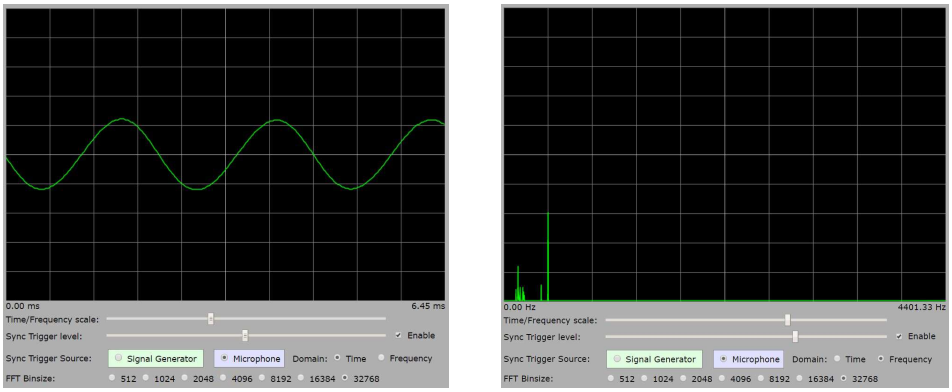
Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

6.1 Timbre

The concept of musical *pitch* is the same as the *fundamental frequency* of that tone. It may be measured in cycles per second (Hertz), or as musicians do by referring to letters (A through G). However, different instruments playing tones having the same pitch nevertheless sound quite different from each other. No one would ever mistake a piano for a violin or for a guitar even at precisely the same pitch (fundamental frequency). So then, why do different instruments sound dissimilar even at the same pitch?

Frequency-domain analysis is very helpful in answering this question. For the purposes of this exercise we will focus on one particular pitch produced by three different instruments. The pitch will be 440 Hertz (A_4 in *scientific pitch notation*). To begin, we will measure[1] the acoustic vibrations produced by a *tuning* fork tuned to this pitch, both in time-domain (left) and frequency-domain (right) formats:

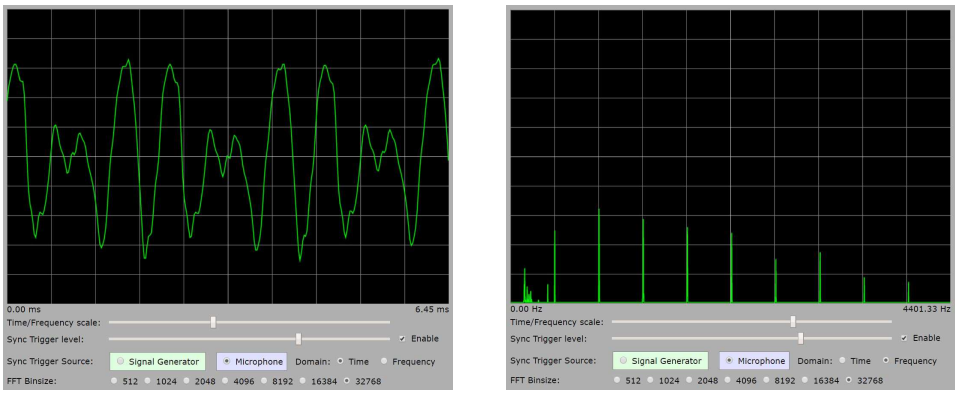


In the time domain the tuning fork's signal looks like a clean sinusoid. In the frequency domain this same signal appears as a solitary peak at 440 Hz[2], with some transient noise present to the left of this peak. This agrees well with theory, as a pure sinusoid should consist of just one peak in the frequency domain. If we listen to the tone of a tuning fork, we will notice it is rather simple and uninteresting.

[1] The software used to generate these oscillograph and spectrum plots is `SigGen` which is a Javascript application running within a web browser. Paul Lutus is the author of this software, with the version shown here being 1.7, copyright 2015.

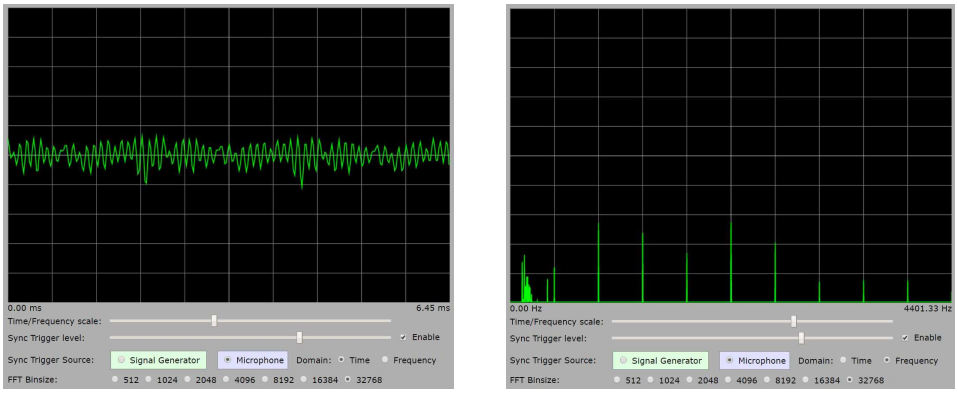
[2] Note the frequency scale on this display, which is linear from 0 Hz (far left) to 4401.33 Hz (far right) over ten divisions, making each division worth 440.133 Hz. This is as close as I could adjust the software's display to a 4400 Hz range.

Next we will analyze the sound produced by a reed instrument, in this case a *melodica* playing the same pitch (440 Hz, A_4):



Here we see a much more complicated wave-shape in the time domain, albeit with the same period (approximately 3.5 divisions on the oscillograph's horizontal axis, or 2.273 milliseconds). The frequency domain plot is also much more complicated than the tuning fork's with harmonics every 440 Hz across the analyzer's display (once again ignoring the transient noise to the left of the first-harmonic or fundamental peak).

After this we will analyze the sound produced by a string instrument, in this case a *bowed psaltery* playing the same pitch (440 Hz, A_4):

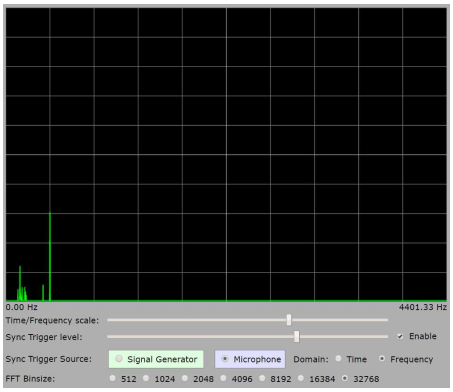


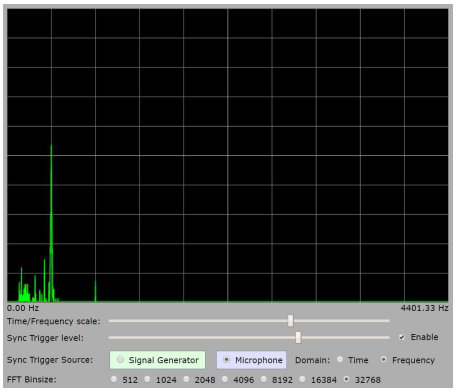
Once again we see a complex wave-shape in the time domain as well as a strong mix of harmonic frequencies in the frequency domain, despite the fundamental frequency still being the same (440 Hz, with a period of approximately 3.5 horizontal divisions) as with the other samples.

It is this mixture of harmonic frequencies, and their strengths relative to each other, that makes each instrument's "voice" unique. The proper musical term for this distinction is *timbre*, alternatively referred to as the *tone quality* or *tone color*. That last descriptor fits well with the Tutorial's analogy of light frequencies being mixed in different proportions to produce different colors.

It is even possible to adjust the timbre of an instrument by playing it differently. Wind instruments such as flutes and trumpets may be driven with stronger flows of air to produce harmonics that are not present when driven softly. Bowed-string instruments may similarly be driven with varying degrees of bow pressure and speed to produce variations in tone quality. One of the marks of an accomplished musician is to be able to reproduce the unique timbres of their instruments on demand.

An elementary example of this is the lowly tuning fork, the spectrum displays of one tuning fork ($A_4 = 440$ Hz) shown side-by-side. The left image shows the sound produced by the tuning fork after being lightly tapped. The image on the right shows the result of striking the tuning fork with greater force:





In addition to greater levels of noise picked up by the analyzer's microphone, as well as a stronger (taller) fundamental peak, we also see a second harmonic peak at 880 Hz due to more complex vibrational modes along the tines of the tuning fork generated by greater striking force.

6.2 Fourier series for common waveforms

The following Fourier series show the harmonic terms only: each fractional coefficient preceding its respective trigonometric function indicates the relative amplitude of each harmonic, while each integer coefficient of ωt indicates the harmonic number. ω is simply frequency in radians per second, and t of course is time. For example, $\frac{1}{9}\cos 3\omega t$ describes the third harmonic in the triangle-wave Fourier series, where that harmonic (having a frequency three times the wave's fundamental) is only one-ninth as intense.

An important omission in these series is the *DC offset*, which is somewhat arbitrary and has no effect whatsoever on the *shape* of the oscillating wave. The goal here is to simply show which sinusoids must be pieced together to form common waveshapes.

6.2.1 Square wave

Square waves consist of an infinite sum of odd harmonics, each harmonic's amplitude being $\frac{1}{n}$ of the fundamental's. Note how all the sinusoids are *sine* functions. This series will produce a square wave with a peak magnitude of exactly one (1):

$$\frac{4}{\pi}\left(\sin\omega t + \frac{1}{3}\sin 3\omega t + \frac{1}{5}\sin 5\omega t + \frac{1}{7}\sin 7\omega t + \cdots + \frac{1}{n}\sin n\omega t\right)$$

6.2.2 Triangle wave

Triangle waves consist of an infinite sum of odd harmonics, each harmonic's amplitude being $\frac{1}{n^2}$ of the fundamental's. Note how all the sinusoids are *cosine* functions. This series will produce a triangle wave with a peak magnitude of exactly one (1):

$$\frac{8}{\pi^2}\left(\cos\omega t + \frac{1}{9}\cos 3\omega t + \frac{1}{25}\cos 5\omega t + \frac{1}{49}\cos 7\omega t + \cdots + \frac{1}{n^2}\cos n\omega t\right)$$

6.2.3 Sawtooth wave

Sawtooth waves are asymmetric about their centerlines, which means they contain even harmonics. In this case, both even and odd harmonics exist in this series, each harmonic's amplitude being $\frac{1}{n}$ of the fundamental's. Note how all the sinusoids are *sine* functions. This series will produce a sawtooth wave with a peak-to-peak magnitude of exactly one (1):

$$\frac{1}{\pi}\left(\sin\omega t + \frac{1}{2}\sin 2\omega t + \frac{1}{3}\sin 3\omega t + \frac{1}{4}\sin 4\omega t + \cdots + \frac{1}{n}\sin n\omega t\right)$$

6.2.4 Pulse wave

A "square" wave with a duty cycle (D) other than 50% is commonly referred to as a *pulse* wave, and will consist of both odd and even harmonics, each harmonic's amplitude being $\frac{1}{n}\sin(n\pi D)$ of the fundamental's, where $n\pi D$ is an angle in radians rather than degrees. Note how all the sinusoids are *cosine* functions, with the sine functions merely serving as coefficients for the magnitude of their respective cosine terms: in other words, each of the sine functions in this series merely sets the peak amplitude of each harmonic, while each of the cosine functions actually creates the sinusoidal shape of each harmonic. This series will produce a square wave with a peak magnitude of exactly one (1):

$$\frac{4}{\pi}\left(\sin(\pi D)\cos(\omega t) + \frac{1}{2}\sin(2\pi D)\cos(2\omega t) + \frac{1}{3}\sin(3\pi D)\cos(3\omega t) + \cdots + \frac{1}{n}\sin(n\pi D)\cos(n\omega t)\right)$$

Note that if you set the duty cycle D to 50% (i.e. $D = 0.5$) then this Fourier Series becomes identical to that of the typical square wave. Another way of saying this is to state that a symmetrical square wave is a *special case* of the pulse waveform. For the first (fundamental) harmonic, $\sin(\pi \times 0.5) = \sin\left(\frac{pi}{2}\right) = 1$; for the second harmonic, $\sin(2 \times \pi \times 0.5) = \sin(\pi) = 0$; etc.

Setting the duty cycle to either 0% or 100% sets all the sine terms to zero, the sine of 0 radians being equal to zero as well as the sine of π radians (or any integer-multiple of π) being equal to zero. The result, of course, will be no waveform at all because every (cosine) harmonic term will have a coefficient of zero. These are other *special cases* of the pulse waveform, showing how a pulse waveform with a duty cycle of 0% or with a duty cycle of 100% contains no alternating (AC) component at all.

Chapter 7

Programming References

A powerful tool for mathematical modeling is text-based *computer programming*. This is where you type coded commands in text form which the computer is able to interpret. Many different text-based languages exist for this purpose, but we will focus here on just two of them, *C++* and *Python*.

7.1 Programming in C++

One of the more popular text-based computer programming languages is called *C++*. This is a *compiled* language, which means you must create a plain-text file containing C++ code using a program called a *text editor*, then execute a software application called a *compiler* to translate your "source code" into instructions directly understandable to the computer. Here is an example of "source code" for a very simple C++ program intended to perform some basic arithmetic operations and print the results to the computer's console:

```
#include <iostream>
using namespace std;

int main (void)
{
  float x, y;

  x = 200;
  y = -560.5;

  cout << "This simple program performs basic arithmetic on" << endl;
  cout << "the two numbers " << x << " and " << y << " and then" << endl;
  cout << "displays the results on the computer's console." << endl;

  cout << endl;

  cout << "Sum = " << x + y << endl;
  cout << "Difference = " << x - y << endl;
  cout << "Product = " << x * y << endl;
  cout << "Quotient of " << x / y << endl;

  return 0;
}
```

Computer languages such as C++ are designed to make sense when read by human programmers. The general order of execution is left-to-right, top-to-bottom just the same as reading any text document written in English. Blank lines, indentation, and other "whitespace" is largely irrelevant in C++ code, and is included only to make the code more pleasing[1] to view.

[1] Although not included in this example, *comments* preceded by double-forward slash characters (`//`) may be added to source code as well to provide explanations of what the code is supposed to do, for the benefit of anyone reading it. The compiler application will ignore all comments.

Let's examine the C++ source code to explain what it means:

- `#include <iostream>` and `using namespace std;` are set-up instructions to the compiler giving it some context in which to interpret your code. The code specific to your task is located between the brace symbols ({ and }, often referred to as "curly-braces").

- `int main (void)` labels the "Main" function for the computer: the instructions within this function (lying between the { and } symbols) it will be commanded to execute. Every complete C++ program contains a `main` function at minimum, and often additional functions as well, but the `main` function is where execution always begins. The `int` declares this function will return an *integer* number value when complete, which helps to explain the purpose of the `return 0;` statement at the end of the `main` function: providing a numerical value of zero at the program's completion as promised by `int`. This returned value is rather incidental to our purpose here, but it is fairly standard practice in C++ programming.

- Grouping symbols such as (parentheses) and {braces} abound in C, C++, and other languages (e.g. Java). Parentheses typically group data to be processed by a function, called *arguments* to that function. Braces surround lines of executable code belonging to a particular function.

- The `float` declaration reserves places in the computer's memory for two *floating-point* variables, in this case the variables' names being `x` and `y`. In most text-based programming languages, variables may be named by single letters or by combinations of letters (e.g. `xyz` would be a single variable).

- The next two lines assign numerical values to the two variables. Note how each line terminates with a semicolon character (`;`) and how this pattern holds true for most of the lines in this program. In C++ semicolons are analogous to periods at the ends of English sentences. This demarcation of each line's end is necessary because C++ ignores whitespace on the page and doesn't "know" otherwise where one line ends and another begins.

- All the other instructions take the form of a `cout` command which prints characters to the "standard output" stream of the computer, which in this case will be text displayed on the console. The double-less-than symbols (`<<`) show data being sent *toward* the `cout` command. Note how verbatim text is enclosed in quotation marks, while variables such as `x` or mathematical expressions such as `x - y` are not enclosed in quotations because we want the computer to display the numerical values represented, not the literal text.

- Standard arithmetic operations (add, subtract, multiply, divide) are represented as `+`, `-`, `*`, and `/`, respectively.

- The `endl` found at the end of every `cout` statement marks the end of a line of text printed to the computer's console display. If not for these `endl` inclusions, the displayed text would resemble a run-on sentence rather than a paragraph. Note the `cout << endl;` line, which does nothing but create a blank line on the screen, for no reason other than esthetics.

After saving this *source code* text to a file with its own name (e.g. `myprogram.cpp`), you would then *compile* the source code into an *executable* file which the computer may then run. If you are using a console-based compiler such as *GCC* (very popular within variants of the Unix operating system[2], such as Linux and Apple's OS X), you would type the following command and press the Enter key:

```
g++ -o myprogram.exe myprogram.cpp
```

This command instructs the `GCC` compiler to take your source code (`myprogram.cpp`) and create with it an executable file named `myprogram.exe`. Simply typing `./myprogram.exe` at the command-line will then execute your program:

```
./myprogram.exe
```

If you are using a graphic-based C++ development system such as Microsoft Visual Studio[3], you may simply create a new console application "project" using this software, then paste or type your code into the example template appearing in the editor window, and finally run your application to test its output.

As this program runs, it displays the following text to the console:

```
This simple program performs basic arithmetic on
the two numbers 200 and -560.5 and then
displays the results on the computer's console.

Sum = -360.5
Difference = 760.5
Product = -112100
Quotient of -0.356824
```

As crude as this example program is, it serves the purpose of showing how easy it is to write and execute simple programs in a computer using the C++ language. As you encounter C++ example programs (shown as source code) in any of these modules, feel free to directly copy-and-paste the source code text into a text editor's screen, then follow the rest of the instructions given here (i.e. save to a file, compile, and finally run your program). You will find that it is generally easier to

[2] A very functional option for users of Microsoft Windows is called *Cygwin*, which provides a Unix-like console environment complete with all the customary utility applications such as GCC!

[3] Using Microsoft Visual Studio community version 2017 at the time of this writing to test this example, here are the steps I needed to follow in order to successfully compile and run a simple program such as this: (1) Start up Visual Studio and select the option to create a New Project; (2) Select the Windows Console Application template, as this will perform necessary set-up steps to generate a console-based program which will save you time and effort as well as avoid simple errors of omission; (3) When the editing screen appears, type or paste the C++ code within the `main()` function provided in the template, deleting the "Hello World" `cout` line that came with the template; (4) Type or paste any preprocessor directives (e.g. `#include` statements, `namespace` statements) necessary for your code that did not come with the template; (5) Lastly, under the Debug drop-down menu choose either Start Debugging (F5 hot-key) or Start Without Debugging (Ctrl-F5 hotkeys) to compile ("Build") and run your new program. Upon execution a console window will appear showing the output of your program.

learn computer programming by closely examining others' example programs and modifying them than it is to write your own programs starting from a blank screen.

7.2 Programming in Python

Another text-based computer programming language called *Python* allows you to type instructions at a terminal prompt and receive immediate results without having to compile that code. This is because Python is an *interpreted* language: a software application called an *interpreter* reads your source code, translates it into computer-understandable instructions, and then executes those instructions in one step.

The following shows what happens on my personal computer when I start up the Python interpreter on my personal computer, by typing `python3`[4] and pressing the Enter key:

```
Python 3.7.2 (default, Feb 19 2019, 18:15:18)
[GCC 4.1.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` symbols represent the prompt within the Python interpreter "shell", signifying readiness to accept Python commands entered by the user.

Shown here is an example of the same arithmetic operations performed on the same quantities, using a Python interpreter. All lines shown preceded by the `>>>` prompt are entries typed by the human programmer, and all lines shown without the `>>>` prompt are responses from the Python interpreter software:

```
>>> x = 200
>>> y = -560.5
>>> x + y
-360.5
>>> x - y
760.5
>>> x * y
-112100.0
>>> x / y
-0.35682426404995538
>>> quit()
```

[4] Using version 3 of Python, which is the latest at the time of this writing.

More advanced mathematical functions are accessible in Python by first entering the line `from math import *` which "imports" these functions from Python's math *library* (with functions identical to those available for the C programming language, and included on any computer with Python installed). Some examples show some of these functions in use, demonstrating how the Python interpreter may be used as a scientific calculator:

```
>>> from math import *
>>> sin(30.0)
-0.98803162409286183
>>> sin(radians(30.0))
0.49999999999999994
>>> pow(2.0, 5.0)
32.0
>>> log10(10000.0)
4.0
>>> e
2.7182818284590451
>>> pi
3.1415926535897931
>>> log(pow(e,6.0))
6.0
>>> asin(0.7071068)
0.78539819000368838
>>> degrees(asin(0.7071068))
45.000001524425265
>>> quit()
```

Note how trigonometric functions assume angles expressed in *radians* rather than *degrees*, and how Python provides convenient functions for translating between the two. Logarithms assume a base of e unless otherwise stated (e.g. the `log10` function for common logarithms).

The interpreted (versus compiled) nature of Python, as well as its relatively simple syntax, makes it a good choice as a person's first programming language. For complex applications, interpreted languages such as Python execute slower than compiled languages such as C++, but for the very simple examples used in these learning modules speed is not a concern.

Another Python math library is `cmath`, giving Python the ability to perform arithmetic on complex numbers. This is very useful for AC circuit analysis using *phasors*[5] as shown in the following example. Here we see Python's interpreter used as a scientific calculator to show series and parallel impedances of a resistor, capacitor, and inductor in a 60 Hz AC circuit:

```
>>> from math import *
>>> from cmath import *
>>> r = complex(400,0)
>>> f = 60.0
>>> xc = 1/(2 * pi * f * 4.7e-6)
>>> zc = complex(0,-xc)
>>> xl = 2 * pi * f * 1.0
>>> zl = complex(0,xl)
>>> r + zc + zl
(400-187.38811239154882j)
>>> 1/(1/r + 1/zc + 1/zl)
(355.837695813625+125.35793777619385j)
>>> polar(r + zc + zl)
(441.717448903332, -0.4381072059213295)
>>> abs(r + zc + zl)
441.717448903332
>>> phase(r + zc + zl)
-0.4381072059213295
>>> degrees(phase(r + zc + zl))
-25.10169387356105
```

When entering a value in rectangular form, we use the `complex()` function where the arguments are the real and imaginary quantities, respectively. If we had opted to enter the impedance values in polar form, we would have used the `rect()` function where the first argument is the magnitude and the second argument is the angle in radians. For example, we could have set the capacitor's impedance (`zc`) as $X_C \angle -90^o$ with the command `zc = rect(xc,radians(-90))` rather than with the command `zc = complex(0,-xc)` and it would have worked the same.

Note how Python defaults to rectangular form for complex quantities. Here we defined a 400 Ohm resistance as a complex value in rectangular form (400 +j0 Ω), then computed capacitive and inductive reactances at 60 Hz and defined each of those as complex (phasor) values ($0 - jX_c$ Ω and $0 + jX_l$ Ω, respectively). After that we computed total impedance in series, then total impedance in parallel. Polar-form representation was then shown for the series impedance (441.717 Ω $\angle -25.102^o$). Note the use of different functions to show the polar-form series impedance value: `polar()` takes the complex quantity and returns its polar magnitude and phase angle in *radians*; `abs()` returns just the polar magnitude; `phase()` returns just the polar angle, once again in radians. To find the polar phase angle in degrees, we nest the `degrees()` and `phase()` functions together.

The utility of Python's interpreter environment as a scientific calculator should be clear from these examples. Not only does it offer a powerful array of mathematical functions, but also unlimited

[5] A "phasor" is a voltage, current, or impedance represented as a complex number, either in rectangular or polar form.

assignment of variables as well as a convenient text record[6] of all calculations performed which may be easily copied and pasted into a text document for archival.

It is also possible to save a set of Python commands to a text file using a text editor application, and then instruct the Python interpreter to execute it at once rather than having to type it line-by-line in the interpreter's shell. For example, consider the following Python program, saved under the filename `myprogram.py`:

```
x = 200
y = -560.5

print("Sum")
print(x + y)

print("Difference")
print(x - y)

print("Product")
print(x * y)

print("Quotient")
print(x / y)
```

As with C++, the interpreter will read this source code from left-to-right, top-to-bottom, just the same as you or I would read a document written in English. Interestingly, whitespace *is* significant in the Python language (unlike C++), but this simple example program makes no use of that.

To execute this Python program, I would need to type `python myprogram.py` and then press the Enter key at my computer console's prompt, at which point it would display the following result:

```
Sum
-360.5
Difference
760.5
Product
-112100.0
Quotient
-0.35682426405
```

As you can see, syntax within the Python programming language is simpler than C++, which is one reason why it is often a preferred language for beginning programmers.

[6] Like many command-line computing environments, Python's interpreter supports "up-arrow" recall of previous entries. This allows quick recall of previously typed commands for editing and re-evaluation.

If you are interested in learning more about computer programming in *any* language, you will find a wide variety of books and free tutorials available on those subjects. Otherwise, feel free to learn by the examples presented in these modules.

7.3 Simple plotting of sinusoidal waves using C++

Like the vast majority of computer programming languages, C and C++ offer an extensive library of mathematical functions ready-made for use in programs of your own design. Here we will examine a C++ program written to calculate the instantaneous values of a sine wave over one full period:

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
    for (n = 0 ; n < (40 * sin(x) + 40) ; ++n)
      cout << " ";

    cout << "*" << endl;
  }

  return 0;
}
```

C++ lacks a standard library of graphics functions for plotting curves and other mathematical shapes to the computer's screen, and so this program instead uses *standard console* characters to do the same. In this particular case it plots blank space characters and star characters (*) to the console in order to mimic a pixel-based graphical display.

This program is deceptively terse. From the small number of lines of code it doesn't look very complicated, but there is a lot going on here. We will explore the operation of this program in stages, first by examining its console output (on the following page), and then analyzing its lines of code.

The result is a somewhat crude, but functional image of a sine wave plotted with amplitude on the horizontal axis and angle on the vertical axis:

Let's analyze how this program works, exploring the following programming principles along the way:

- Order of execution
- Preprocessor directives, namespaces
- The `main` function: return values, arguments
- Delimiter characters (e.g. `{ } ;`)
- Whitespace ignored
- Variable types (`float` and `int`), names, and declarations
- Variable assignment/initialization (`=`)
- Comparison (`==`)
- Loops (`for`)
- Incrementing variables (`++`)
- Basic arithmetic (`+`, `*`)
- Arithmetic functions (`sin`)
- Printing text output (`cout`)
- Comments (`//`)
- Custom functions: prototyping, return values, arguments

Looking at the source code listing, we see the obligatory[7] directive lines at the very beginning (`#include` and `namespace`) telling the C++ compiler software how to interpret many of the instructions that follow. Also obligatory for any C++ program is the `main` function enclosing all of our simulation code. The line reading `int main (void)` tells us the `main` function takes in no data (`void`) but returns an integer number value (`int`). The "left-curly-brace" symbol immediately below that ({) marks the beginning of the page space where the `main` function's code is found, while the "right-curly-brace" symbol at the bottom (}) marks the end of the `main` function. All code located between those brace symbols belongs to the `main` function. All indentation of lines is done merely to make the source code easier for human eyes to read, and not for the sake of the C++ compiler software which ignores whitespace.

Within the `main` function we have two variables declared, two `for` instructions, and two `cout` statements. Variable `x` is a floating-point variable, intended to store the angle values we will send to the sine function. Variable `n` is an integer variable, capable only of counting in whole-number steps. A `for` loop instructs the computer to repeat some operation multiple times, the number of repeats determined by the value of some variable within the `for` instruction's parentheses.

[7] The `#include <iostream>` directive is necessary for using standard input/output instructions such as `cout`. The `#include <cmath>` directive is necessary for using advanced mathematical functions such as sine.

Our first `for` instruction bases its repeats on the value of `x`, beginning by initializing it to a value of zero and then incrementing it in steps of 0.2 so long as `x` is less than or equal to 2π. This `for` loop has its own set of "curly-brace" symbols enclosing multiple lines of code, again with those lines indented to make it visually clear they belong within the `for` loop.

Within this outer `for` loop lies another `for` instruction, with its repeats based on the value of our integer variable `n`. Unlike the outer `for` loop which has brace symbols ({}) enclosing multiple lines of code, the inner `for` loop has no braces of its own because only one line of code belongs to it (a `cout` instruction printing blank spaces to the console, found immediately below the `for` statement and indented to make its ownership visually clear). This inner `for` instruction's repeats continue so long as `n` remains less than the value of $40\sin(x) + 40$, incrementing from 0 upwards in whole-number steps (this is what `++n` means in the C and C++ languages: to increment an integer variable by a single-step). Below that is another `cout` instruction, this one printing a star character to the console (`*`).

It may not be clear to the reader how these two `for` instructions work together to create a sinusoidal pattern of characters on the computer's console display, and so we will spend some more time dissecting the code. A useful problem-solving strategy for understanding this program is to *simplify the system*. In this case we will replace all lines of code within the outer `for` loop with a single `cout` instruction printing values of `x` and `sin(x)`. This will generate a listing of these variables' values, which as we know governs the two `for` loops' behavior. Once we see these numerical values, it will become easier to grasp what the `for` loops and their associated `cout` instructions are trying to achieve.

Rather than *delete* the original lines of code, which would require re-typing them at some point in the future, we will apply a common programming "trick" of *commenting out* those lines we don't want to be executed. In C and C++, and double-forward-slash (//) marks the beginning of an inline comment, with all characters to the right of the double-slashes ignored by the compiler. They will still be in the source code, readable to any human eyes, but will be absent from the program as far as the computer is concerned. Then, when we're ready to reinstate these code lines again, all we need to do is delete the comment symbols:

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
//    for (n = 0 ; n < (40 * sin(x) + 40) ; ++n)
//      cout << " ";

//    cout << "*" << endl;
    cout << x << "     " << sin(x) << endl;
  }

  return 0;
}
```

Re-compiling the modified code and re-running it produces the following results:

```
0     0
0.2     0.198669
0.4     0.389418
0.6     0.564642
0.8     0.717356
1     0.841471
1.2     0.932039
1.4     0.98545
1.6     0.999574
1.8     0.973848
2     0.909297
2.2     0.808496
2.4     0.675463
```

```
2.6     0.515501
2.8     0.334988
3     0.14112
3.2     -0.0583747
3.4     -0.255542
3.6     -0.442521
3.8     -0.611858
4     -0.756803
4.2     -0.871576
4.4     -0.951602
4.6     -0.993691
4.8     -0.996165
5     -0.958924
5.2     -0.883455
5.4     -0.772765
5.6     -0.631267
5.8     -0.464603
6     -0.279417
6.2     -0.083091
```

Not surprisingly, we see the variable `x` increment from zero to 6.2 (approximately 2π) in steps of 0.2. The sine of this angle value evolves from 0 to very nearly +1, back (almost) to zero as `x` goes past π, very nearly equaling -1, and finally returning close to zero. This is what we would expect of the trigonometric *sine* function with its angle expressed in *radians* rather than degrees (2π radians being equal to 360 degrees, a full circle).

This experiment proves to us what `x` and `sin(x)` are doing in the program, but to more clearly see how the inner `for` loop functions it would be helpful to print the value of `40 * sin(x) + 40` since this is the actual value checked by the inner `for` loop as it increments `n` from zero upward.

Modifying the code once more for another experiment:

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
//    for (n = 0 ; n < (40 * sin(x) + 40) ; ++n)
//      cout << " ";

//    cout << "*" << endl;
    cout << x << "     " << 40 * sin(x) + 40 << endl;
  }

  return 0;
}
```

Re-compiling this new code and running it reveals much larger values in the second number column:

```
0     40
0.2     47.9468
0.4     55.5767
0.6     62.5857
0.8     68.6942
1     73.6588
1.2     77.2816
1.4     79.418
1.6     79.9829
1.8     78.9539
2     76.3719
2.2     72.3399
2.4     67.0185
2.6     60.62
2.8     53.3995
3     45.6448
3.2     37.665
3.4     29.7783
```

```
3.6     22.2992
3.8     15.5257
4     9.72789
4.2     5.13696
4.4     1.93592
4.6     0.252361
4.8     0.153415
5     1.64302
5.2     4.6618
5.4     9.0894
5.6     14.7493
5.8     21.4159
6     28.8233
6.2     36.6764
```

Instead of progressing from zero to (nearly) +1 to (nearly) zero to (nearly) −1 and back again to (nearly) zero, this time the right-hand column of numbers begins at 40, progresses to a value of (nearly) 80, then back past 40 and (nearly) to zero, then finishes nearly at 40 again. What the `40 * sin(x) + 40` arithmetic[8] does is "scale" and "shift" the basic sine function to have a peak value of 40 and a center value of 40 as well.

[8] You may recognize this as the common slope-intercept form of a linear equation, $y = mx + b$. In this case, 40 is the slope (m) and 40 also happens to be the intercept (b).

Now that we clearly recognize the range of `40 * sin(x) + 40`, we may remove the comments from our code and analyze the inner `for` loop:

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
    for (n = 0 ; n < (40 * sin(x) + 40) ; ++n)
      cout << " ";

    cout << "*" << endl;
  }

  return 0;
}
```

Each time the outer `for` loop increments the value of x, the inner `for` loop calculates the value of $40\sin(x) + 40$ and repeats the `cout << " "` instruction that many times[9] to print that same number of blank spaces on the console. After printing that string of blank spaces, the second `cout` statement prints a star character (`*`) and finishes the line with an `endl` character (a carriage-return marking the end of a line and the beginning of a new line on the console's display). The outer `for` loop then increments x again and the process repeats.

Therefore, the outer `for` loop produces one new line of text on the console per iteration, while the inner `for` loop produces one new blank space on that line per iteration. This makes the placement of each star character (`*`) proportional to the value of $\sin(x)$, the result being a "sideways" plot of a sine wave on the console.

The scaling of the sine function to produce a range from 0 to +80 rather than −1 to +1 was intentionally chosen to fit the standard 80-column width of traditional character-based computer consoles. Modern computer operating systems usually provide *terminal* windows emulating traditional consoles, but with font options for resizing characters to yield more or less than 80 columns spanning the console's width.

[9] The value of $40\sin(x) + 40$ will be a floating-point (i.e. non-round) value, while `n` is an integer variable and can therefore only accept whole-numbered (and negative) values. This is not a problem in C++, as the compiler is smart enough to cause the floating-point value to become truncated to an integer value before assigning it to `n`.

Students more accustomed to applied trigonometry than pure mathematics may bristle at the assumed unit of *radians* used by C++ when computing the sine function, but this is actually quite common for computer-based calculations. Even most electronic hand calculators assume radians unless and until the user sets the *degree* mode.

We can modify this code have the variable `x` in degrees rather than radians, simply by multiplying `x` by the conversion factor $\frac{\pi}{180}$.

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= 360 ; x = x + 12.0)
  {
    for (n = 0 ; n < (40 * sin(x * (M_PI / 180)) + 40) ; ++n)
      cout << " ";

    cout << "*" << endl;
  }

  return 0;
}
```

This is a good illustration of how mathematical operations may be "nested" within sets of parentheses, in the same way we do so when writing regular formulae:

$$40 \sin \left[x \left(\frac{\pi}{180} \right) \right] + 40$$

An extremely important computer programming concept we may apply at this juncture, though by no means necessary for this simple program, is to include our own custom *function* to calculate the scaled sine value with its degrees-to-radians conversion. The idea of a programming "function" is a separate listing of code lying outside of the `main` function which may be invoked at any time within the `main` function. Some legacy programming languages such as FORTRAN and Pascal referred to these as *subroutines*.

Consider the following version of the sine-plotting program with a custom function called `sinecalc`:

```
#include <iostream>
#include <cmath>
using namespace std;

float sinecalc (float);

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= 360 ; x = x + 12.0)
  {
    for (n = 0 ; n < sinecalc(x) ; ++n)
      cout << " ";

    cout << "*" << endl;
  }

  return 0;
}

float sinecalc (float degrees)
{
  float radians;

  radians = degrees * M_PI / 180;

  return 40 * sin (radians) + 40;
}
```

Note in particular these three alterations made to the code:

- The inclusion of a line before the `main` function *prototyping* our custom function, declaring it will accept a single floating-point value and return a floating-point value.
- The inner `for` statement is much simpler than before without all the inline arithmetic. Now it simply "calls" the `sinecalc` function every time it needs to compute the sine of `x`.
- Past the end of the `main` function is where our new `sinecalc` function resides. Like the `main` function itself, it begins with a line stating it will accept a single floating-point variable (named

`degrees`) and will return a floating-point value. Also like the `main` function, it has its own set of curly-brace symbols ({ }) to enclose its lines of code.

Within the `sinecalc` function we see an declaration of another variable named `radians`, an arithmetic statement performing the degrees-to-radians conversion, and finally a `return` statement where the scaled sine value is computed. This returned value is what the `for` statement "sees" after calling the `sinecalc` function.

The path of a program's execution is no longer simply left-to-right and top-to-bottom once we start using our own functions like this. Now the execution path *jumps* from one line to another and then *returns* back where it left off. This new pattern of execution may seem strange and confusing, but it actually makes larger programs easier to manage and design. By encapsulating a particular algorithm (i.e. a set of instructions and procedures) in its own segment of code separate from the `main` function, we make the `main` function's code more compact and easier to understand. It is even possible to save these functions' code in separate source files so that different human programmers can work on pieces of the whole program separately as a team[10].

[10] For example, we could save all the `main` function's code (including the directive lines) to a file named `main.cpp`, then do the same with the `sinecalc` function's code (also including the necessary directive lines) in a file named `sine.cpp`. The command we would then use to compile and link these two code sets together into an executable named `plot.exe` would be `g++ -o plot.exe main.cpp sine.cpp`.

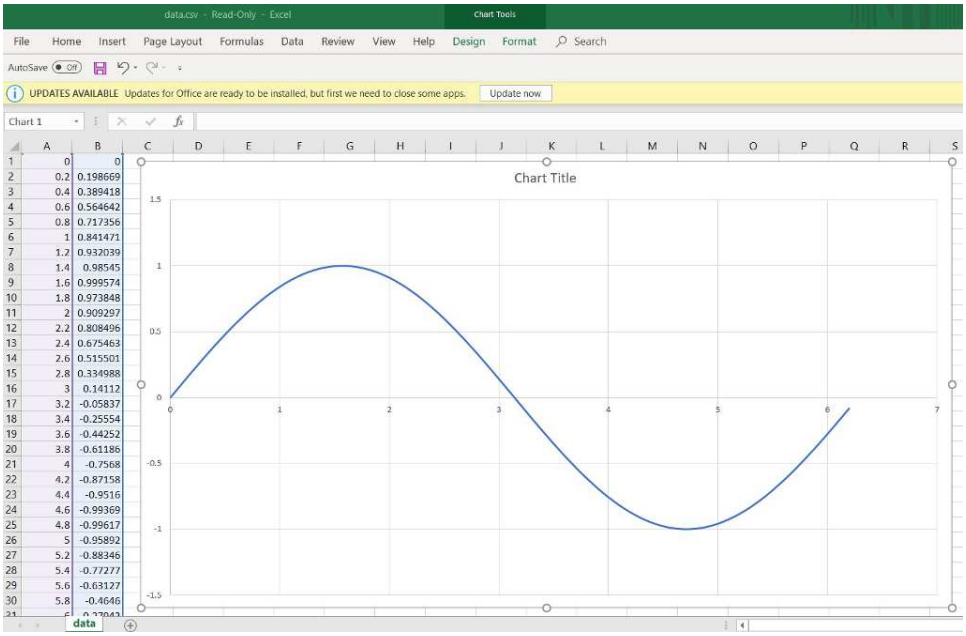
As previously mentioned, C++ lacks a standard library of graphics functions for plotting curves and other mathematical shapes to the computer's screen, which is why we opted to use *standard console* characters to do the same. If a truly *graphic* output is desired for our waveform plot, there are relatively simple alternatives. One is to write the C++ source code to output data as numerical values displayed in columns, one column of numbers representing independent (x) values and the other column representing dependent (y) values, with each column separated by a comma character (,) as a *delimiter*. Here is the re-written program and its text output:

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float x, y;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
    y = sin(x);
    cout << x << "," << y << endl;
  }

  return 0;
}
```

```
0,0
0.2,0.198669
0.4,0.389418
0.6,0.564642
0.8,0.717356
1,0.841471
1.2,0.932039
1.4,0.98545
1.6,0.999574
1.8,0.973848
2,0.909297
2.2,0.808496
2.4,0.675463
2.6,0.515501
2.8,0.334988
3,0.14112
3.2,-0.0583747
3.4,-0.255542
3.6,-0.442521
```

```
3.8,-0.611858
4,-0.756803
4.2,-0.871576
4.4,-0.951602
4.6,-0.993691
4.8,-0.996165
5,-0.958924
5.2,-0.883455
5.4,-0.772765
5.6,-0.631267
5.8,-0.464603
6,-0.279417
6.2,-0.083091
```

We may save this text output to its own file (e.g. `data.csv`)[11] and then import that file into a graphing program such as a spreadsheet (e.g. Microsoft Excel). Spreadsheet software is designed to accept comma-separated variable (`csv`) data and automatically organize the values into columns and rows. Since spreadsheet software is so readily available, this is an easy option to visualize any C++ program's data without having to write C++ code directly generating graphic images.

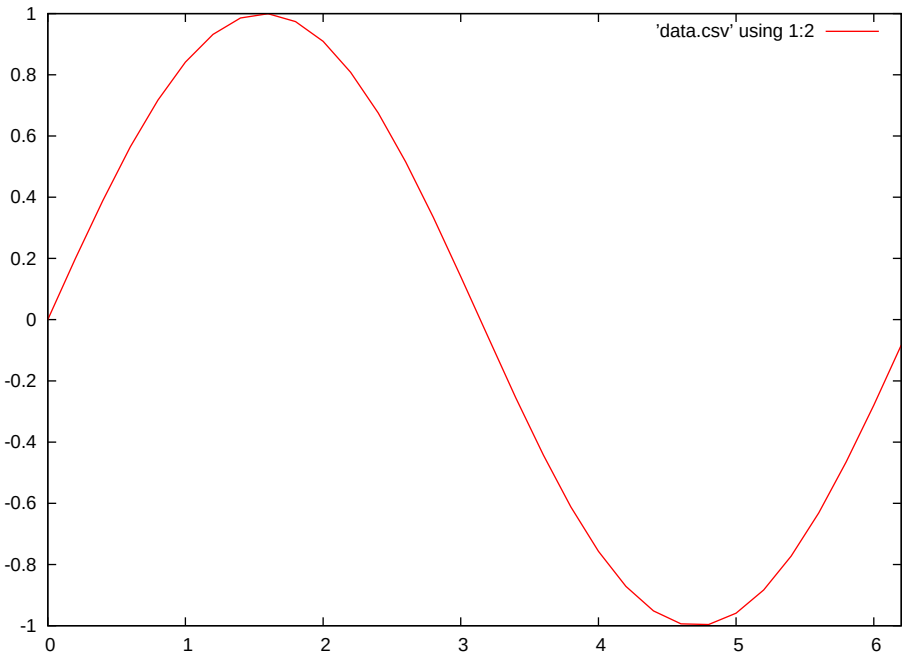


[11] A relatively easy way to do this is to run the C++ program from a console, using the *redirection* symbol (`>`). For example, if we saved our source code file under the name `sinewave.cpp` and then entered `g++ -o sinewave.exe sinewave.cpp` at the command-line interface to compile it, the resulting executable file would be named `sinewave.exe`. If we simply type `./sinewave.exe` and press Enter, the program will run as usual. If, however we type `./sinewave.exe > data.csv` and press Enter, the program will run "silently" with all of its printed text output redirected into a file named `data.csv` instead of to the console for us to see.

Spreadsheets are not the only data-visualizing tools available, though. One such alternative is the open-source software application called `gnuplot`. The following example shows how `gnuplot` may be instructed[12] to read a comma-separated variable file (`data.csv`) and plot that data to the computer's screen:

`gnuplot` script:

```
set datafile separator ","
set xrange [0:6.2]
set style line 1 lw 2 lc rgb "red"
plot 'data.csv' using 1:2 with lines ls 1
```



[12] These commands may be entered interactively at the `gnuplot` prompt or saved to a text file (e.g. `format.txt`, called a *script*) and invoked at the operating system command line (e.g. `gnuplot -p format.txt`).

7.4 Plotting two sinusoidal waves with phase angles using C++

Here we will examine a C++ program written to take input from the user and generate comma-separated value lists for two sinusoidal waveforms which may be plotted using graphical visualization software such as a spreadsheet or `gnuplot`:

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void)
{
  float va, vb, pa, pb, f, period, t;

  cout << "Enter peak amplitude of voltage A" << endl;
  cin >> va;

  cout << "Enter phase angle of voltage A" << endl;
  cin >> pa;

  cout << "Enter peak amplitude of voltage B" << endl;
  cin >> vb;

  cout << "Enter phase angle of voltage B" << endl;
  cin >> pb;

  cout << "Enter frequency for both sources" << endl;
  cin >> f;

  period = 1/f;

  for (t = 0 ; t <= (2 * period) ; t = t + (period/100))
  {
    cout << t << " , " ;
    cout << va * sin((t * f + (pa/360)) * 2 * M_PI) << " , ";
    cout << vb * sin((t * f + (pb/360)) * 2 * M_PI) << endl;
  }

  return 0;
}
```

Let's analyze how this program works, exploring the following programming principles along the way:

- Order of execution
- Preprocessor directives, namespaces
- The `main` function: return values, arguments
- Delimiter characters (e.g. { } ;)
- Whitespace ignored
- Variable types (`float`), names, and declarations
- Variable assignment/initialization (`cin`)
- Loops (`for`)
- Incrementing variables (`++`)
- Basic arithmetic (`+`, `*`)
- Arithmetic functions (`sin`)
- Printing text output (`cout`)

Looking at the source code listing, we see the obligatory[13] directive lines at the very beginning (`#include` and `namespace`) telling the C++ compiler software how to interpret many of the instructions that follow. Also obligatory for any C++ program is the `main` function enclosing all of our simulation code. The line reading `int main (void)` tells us the `main` function takes in no data (`void`) but returns an integer number value (`int`). The "left-curly-brace" symbol immediately below that ({) marks the beginning of the page space where the `main` function's code is found, while the "right-curly-brace" symbol at the bottom (}) marks the end of the `main` function. All code located between those brace symbols belongs to the `main` function. All indentation of lines is done merely to make the source code easier for human eyes to read, and not for the sake of the C++ compiler software which ignores whitespace.

Within the `main` function we have seven variables declared, all of them floating-point (`float`) variables. Several `cout` statements print text to the screen while `cin` statements receive typed input from the user to initialize the values of five of those variables. Variable `t` represents *time*, and is stepped in value from zero to two full periods of the waveforms within the `for` loop. Within the curly-brace symbols of the `for` loop we have a set of `cout` instructions which print the comma-separated value data to the computer's console.

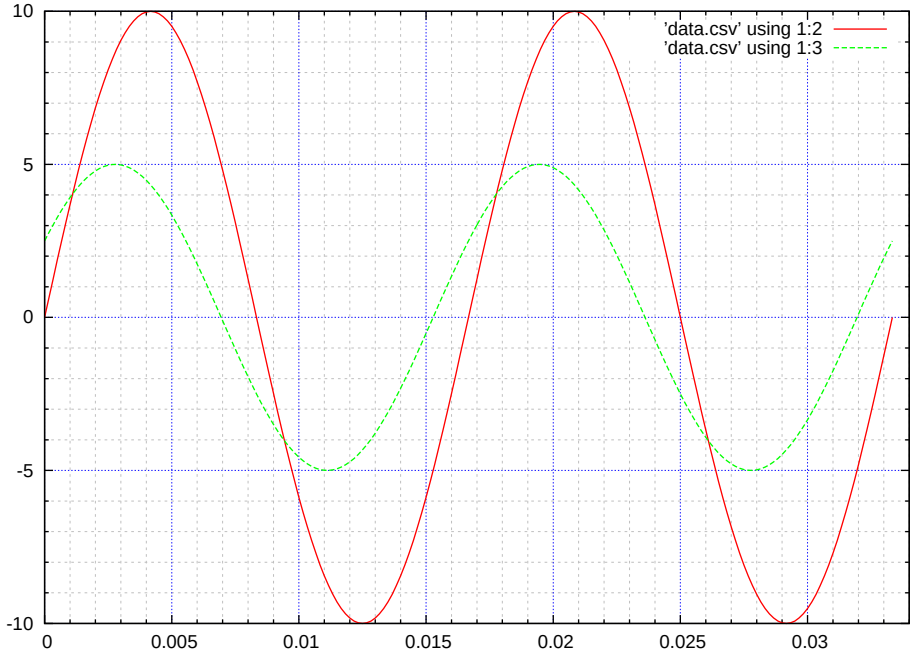
The sine functions are computed within these last `cout` instructions. The product of time and frequency (*seconds* times *cycles per second*) yields a result in *cycles*. Phase shift was entered in *degrees*, so division by 360 is necessary to case phase shift into cycles because there are 360 degrees per cycle. The sine function, like all trigonometric functions in computer programming, requires an input in units of *radians* which explains the purpose of the 2π multiplier, there being 2π radians per cycle.

[13] The `#include <iostream>` directive is necessary for using standard input/output instructions such as `cout`. The `#include <cmath>` directive is necessary for using the sine function.

Here is a sample run of this program where waveform A is 10 Volts (peak) at an angle of 0 degrees and waveform B is 5 Volts (peak) with a leading phase shift of 30 degrees, both at a frequency of 60 Hz:

```
Enter peak amplitude of voltage A 10
Enter phase angle of voltage A 0
Enter peak amplitude of voltage B 5
Enter phase angle of voltage B 30
Enter frequency for both sources 60
0 , 0 , 2.5
0.000166667 , 0.627905 , 2.76696
0.000333333 , 1.25333 , 3.023
0.0005 , 1.87381 , 3.2671
0.000666667 , 2.4869 , 3.49832
0.000833333 , 3.09017 , 3.71572
0.001 , 3.68125 , 3.91847
0.00116667 , 4.25779 , 4.10575
0.00133333 , 4.81754 , 4.27682
```

The comma-separated value list has been shortened for the sake of brevity (from approximately 200 lines of data). When copied to a plain-text file named `data.csv` and read by a data visualization program (in this case, `gnuplot`), the two sinusoids with their differing amplitudes and 30 degree phase shift are clear to see. Setting the visualization tool to show four minor divisions in between every major division mimics the graticule of a traditional oscilloscope:



Any data visualization tool capable of reading a comma-separated value data file is fine for this purpose, and a spreadsheet such as Microsoft Excel is probably the simplest one to use. My favorite happens to be `gnuplot`, and the script I used to make the previous plot is as follows:

```
set datafile separator ","
set xrange [0:0.034]
set style line 1 lw 2 lc rgb "red"
set style line 2 lw 2 lc rgb "green"
set style line 3 lw 0.25 lc rgb "grey"
set style line 4 lw 0.5 lc rgb "blue"
set mxtics 5
set mytics 5
set grid xtics mxtics ls 4, ls 3
set grid ytics mytics ls 4, ls 3
plot 'data.csv' using 1:2 with lines ls 1, 'data.csv' using 1:3 with lines ls 2
```

A simple way to copy the comma-separated value data into the `data.csv` file when running the compiled C++ program is to use the `tee` operator available on the command-line interface of the computer's operating system. Assuming our compiled C++ program is named `phaseplot.exe`, the command-line instruction would look something like the following:

```
phaseplot.exe | tee data.csv
```

This records *all* text – including the prompts for the user's input as well as the entries – into the `data.csv` which must be deleted prior to reading by the spreadsheet or other visualization software. However, some may find the deletion of those few lines easier than the copying-and-pasting of 200+ lines of data to a file.

7.5 Plotting harmonic series using C++

Building on the success of the C++ program from the previous section, we may extend the program's capability to plotting *harmonic series* (i.e. a sum of sinusoids having integer-multiple frequencies of the fundamental).

```
#include <iostream>
#include <cmath>
using namespace std;

float sinecalc (float);

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= 360 ; x = x + 12.0)
  {
    for (n = 0 ; n < (sinecalc(x) + (sinecalc(3.0*x) / 3.0)
        + (sinecalc(5.0*x) / 5.0) + (sinecalc(7.0*x) / 7.0)
        + (sinecalc(9.0*x) / 9.0) + (sinecalc(11.0*x) / 11.0)
        + 40) ; ++n)
      cout << " ";

    cout << "*" << endl;
  }

  return 0;
}

float sinecalc (float degrees)
{
  float radians;

  radians = degrees * M_PI / 180;

  return 40 * sin (radians);
}
```

Note the very long `for` statement, lengthy because the waveform calculation term consists of the fundamental, 3rd harmonic, 5th harmonic, 7th harmonic, 9th harmonic, and 11th harmonic added together, each harmonic with an amplitude reduced by the same factor. This is the recipe for synthesizing a square wave.

When we compile and run this program, we obtain a wave-shape that somewhat resembles a square wave:

The lengthy `for` statement in this program stands as an excellent example of how C++ ignores whitespace in the source code. Portions of the `for` statement are written on successive lines of the source file, but the compiler still interprets those fragments as one long line of code.

A subtle yet important adaptation made to this program from its previous (single sine-wave) version is to move the +40 offset quantity from the `sinecalc` function to the `for` statement in the `main` function. If this offset value were left in the `sinecalc` function, *every harmonic* would add its own +40 offset to the series, the result being that our simple console-based plotting function would quickly run off the page when it tried to print the star symbols (`*`) representing the composite wave.

It is possible to achieve a more perfect square wave by including more harmonics in our series, because the theoretical Fourier series for a square wave is *infinitely long*. However, the way our program is presently designed we would have to lengthen the `for` statement even further with additional `sinecalc` terms to model this. At some point our source code is going to become impractically large.

Computers are very good at executing *repeated tasks*, it should be possible to add code to our program instructing it to repeatedly adding more harmonic terms to the series according to a formula rather than us typing in each term by hand in the source code. One possible solution is shown on the following pages, consisting of the `main` function as well as one more function in addition to `sinecalc` (called `squareseries`):

```
#include <iostream>
#include <cmath>
using namespace std;

float sinecalc (float);
float squareseries (float,int);

int main (void)
{
  float x;
  int n, odds;

  cout << "How many odd harmonics do you wish to add to the fundamental? ";
  cin >> odds;

  for (x = 0 ; x <= 360 ; x = x + 12.0)
  {
    for (n = 0 ; n < (squareseries(x,odds) + 40) ; ++n)
      cout << " ";

    cout << "*" << endl;
  }

  return 0;
}

float sinecalc (float degrees)
{
  float radians;

  radians = degrees * M_PI / 180;

  return 40 * sin (radians);
```

```
}

float squareseries (float angle, int harmonics)
{
  int n;
  float sum;

  sum = 0.0;

  for (n = 0 ; n <= harmonics ; ++n)
    sum = sum + (sinecalc((2 * n + 1) * angle) / (2 * n + 1));

  return sum;
}
```

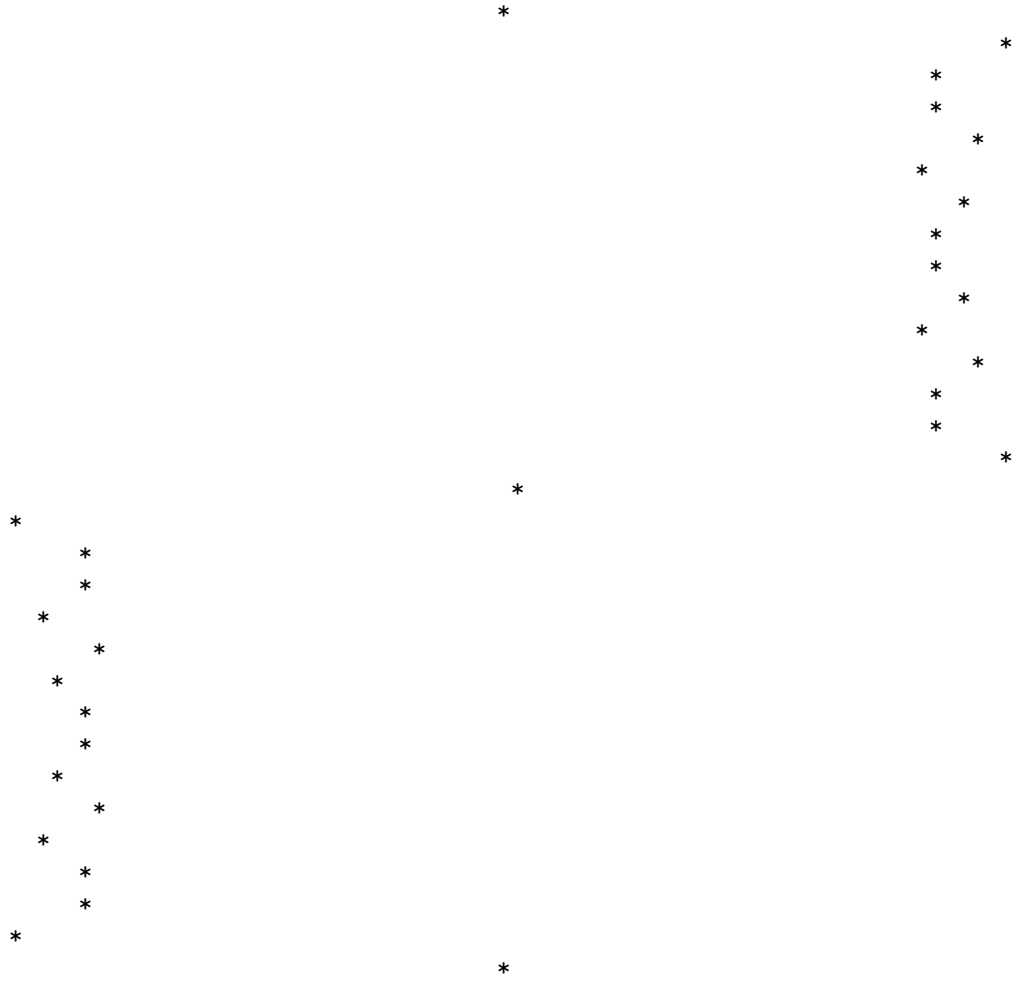
This new function `squareseries` returns a floating-point value just like `sinecalc`, but it accepts *two variables* as arguments: one of them floating-point and the other an integer. The floating-point argument to `squareseries` is the angle value sent to it by the `main` function's inner `for` loop, while the integer argument tells it how many odd-numbered harmonics to add to the fundamental.

With three functions in total (`main`, `sinecalc`, and `squareseries`), we learn some important facts about C/C++ functions:

- All functions should be prototyped, as shown before the `main` function listing.
- Functions get to rename the values passed on to them, as we see in the case of `squareseries`: when called within `main` its two arguments come from the variables `x` and `odds`, but when executing it knows these two values by variable-names local to the `squareseries` function (`angle` and `harmonics`).
- Identically-named functions within different functions are distinct from each other. For example, `n` used within the `main` function counts the number of blank spaces to print to the console, while `n` used within the `squareseries` function sets the coefficient for each harmonic.
- Functions can call other functions! Note how `squareseries` is called from within `main`, and in turn `squareseries` repeatedly calls `sinecalc`.

To test this program, we will first run it with the same number of harmonics in the series as before (harmonics 1, 3, 5, 7, 9, and 11 represent the fundamental plus *five* additional harmonics):

```
How many odd harmonics do you wish to add to the fundamental? 5
```



True to form, we obtain the same square-ish looking wave as before. This is good evidence that our new program is working as it should.

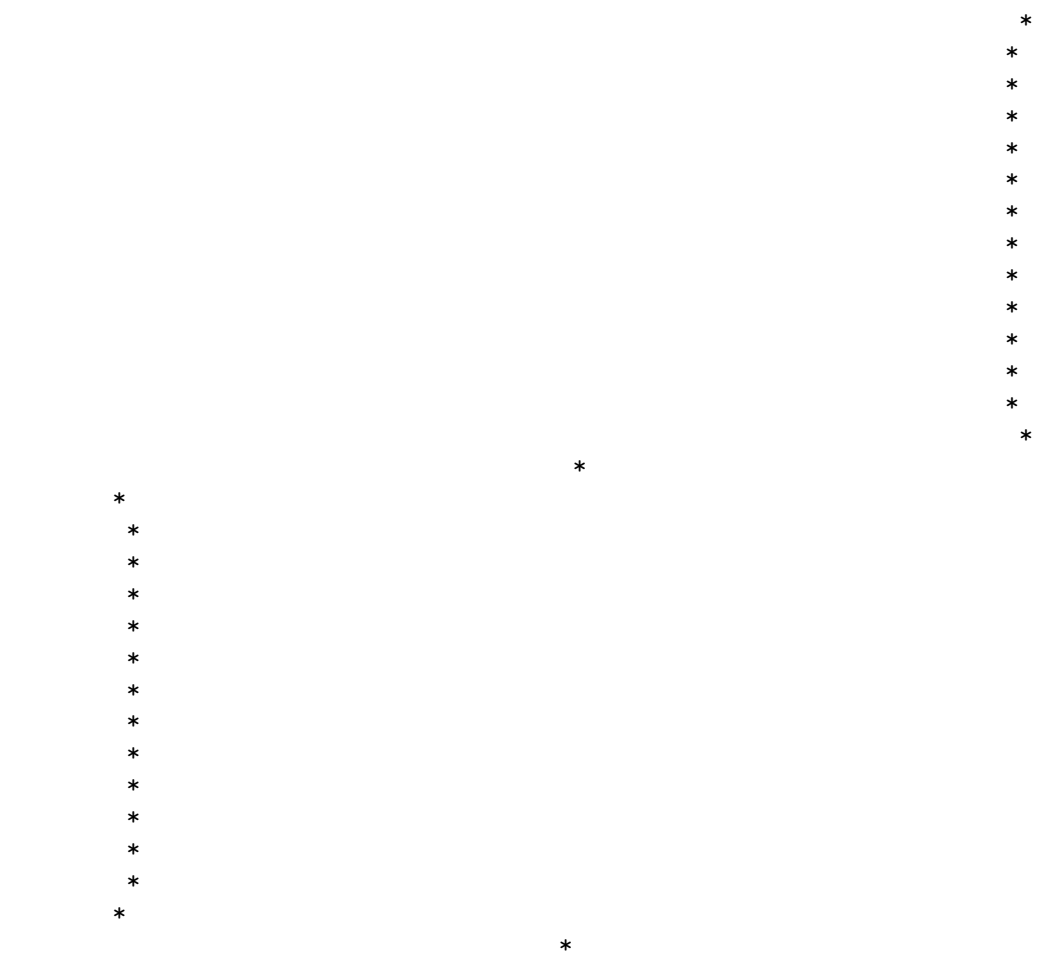
Now let's re-run the program using a much larger harmonic series, with *fifteen* harmonic terms beyond the fundamental:

```
How many odd harmonics do you wish to add to the fundamental? 15
```

This waveform looks *much* closer in shape to a true square wave!

Do we dare try more harmonic terms? Lets!

```
How many odd harmonics do you wish to add to the fundamental? 50
```



Do I even need to ask?

```
How many odd harmonics do you wish to add to the fundamental? 100
                                        *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                                                *
                                        *
            *
            *
            *
            *
            *
            *
            *
            *
            *
            *
            *
            *
            *
                                        *
```

At 100 harmonics past the fundamental (i.e. including all odd-numbered harmonics through the 201^{st}), the waveform is so close to being a perfect square wave that the resolution of our crude text-character plot cannot show the imperfections.

7.6 Discrete Fourier Transform algorithm in C++

The following page of C++ code is the `main()` function for a Discrete Fourier Transform algorithm. As written, this C++ program simulates a square wave and computes the DC average value as well as the first nine harmonics of this wave, although the `f(x)` function code could be re-written to generate any test waveform desired.

A DFT algorithm requires no calculus, only simple trigonometric functions (sine and cosine) and basic arithmetic (multiplication and addition, squares and square roots). The basic idea of it is simple enough: multiply the instantaneous values of the test waveform by the corresponding values of a sinusoid at some harmonic of the test frequency, and sum all of those values over one period of the test waveform. If the sum adds up to zero (or nearly) zero, then that harmonic does not exist in the test waveform. The magnitude of this sum indicates how strong the harmonic is in the test waveform.

Even the mathematical foundation of the DFT is simple, and requires no calculus. It is based on trigonometric identities, specifically those involving the product (multiplication) of sine and/or cosine terms. When two sinusoids of differing frequency are multiplied together, the result is two completely different sinusoids: one having a frequency equal to the sum of the two original frequencies, and the other having a frequency equal to the difference of the two original frequencies. The basic trigonometric identity is shown here:

$$\cos x \cos y = \frac{\cos(x - y) + \cos(x + y)}{2}$$

Next, is the version of this using ω_x and ω_y to represent the two waves' frequencies:

$$\cos(\omega_x t) \cos(\omega_y t) = \frac{\cos(\omega_x t - \omega_y t) + \cos(\omega_x t + \omega_y t)}{2}$$

If the sinusoids being multiplied happen to have the same frequency and be in-phase with each other, the result is a second harmonic and a DC (constant) value (i.e. one sinusoid having a frequency of 2ω and the other having a frequency of zero). So, in order to test a waveform for the presence of a particular harmonic, we multiply it by that other harmonic and see if the resulting product contains DC. How do we test a wave for DC? We sum up all its instantaneous values and see if the result is anything other than zero!

Any practical DFT needs to be just a bit more sophisticated, though, because we must account for phase. We obtain a DC-containing product only if the frequencies *and* phases match. If we happen to multiply a wave by another that's exactly 90^o out of phase, we don't get any DC. To account for phase shift, then, what we do is compute two products and two sums: one based on a sine wave and the other based on a cosine wave (i.e. 90^o apart from each other, so at least one of these two sums will show a match) and then tally their respective sums by the Pythagorean theorem: $\sqrt{x^2 + y^2}$. The rationale for using sine and cosine waves is the same as representing an AC phasor quantity in rectangular form: the sum based on cosines represents the real component of the phasor while the sum based on sines represents the imaginary component. $\sqrt{x^2 + y^2}$ simply computes the polar-form magnitude of these sinusoids' sums.

```
#include <iostream>
#include <math.h>
using namespace std;

float f(int x);

int main(void)
{
  int sample, harmonic;
  float sinsum, cossum, polarsum[10];

  for (harmonic = 1; harmonic < 10; ++harmonic)
    {
      sinsum = 0;
      cossum = 0;

      for (sample = 0; sample < 128; ++sample)
        {
          sinsum = sinsum + (f(sample) * (sin(sample*harmonic*2*M_PI/128)));
          cossum = cossum + (f(sample) * (cos(sample*harmonic*2*M_PI/128)));
        }

      polarsum[harmonic] = sqrt(pow(cossum, 2) + pow(sinsum, 2));

      cout << "Harmonic = " << harmonic << " -- Normalized weight = "
      << fixed << polarsum[harmonic] / polarsum[1] << endl;
    }

  return 0;
}

float f(int x)
{
  if (x < 64)
    return 1.0;

  else
    return -1.0;
}
```

What follows is an explanation of how this DFT algorithm's code works.

- The `include` and `namespace` directives instruct the compiler to be prepared for functions of text printing (`iostream`) and for mathematics (`math.h`).
- The next line (`float f(int x);`) is a *function prototype* for a C++ function named `f`. This function purposely resembles the standard mathematical function form $f(x)$ because it is where the code will reside for the waveform to be analyzed. The input to this function will be an integer number, and the output will be a floating-point number (i.e. capable of fractional values, unlike an integer). The domain of our function happens to be 0 to 127, in whole-numbered steps. The range of our function can be anything representable by a floating-point number. The actual code for this function may appear later in the file (as is the case in this example), or it may even reside in its own source file to be linked to the main program at compilation time.
- Inside the `main()` function we first declare several variables, both integer and floating-point. All mathematical functions are computed over 128 samples, numbered 0 through 127. During each of these samples, we compute the value of our test waveform (`f(x)`) and multiply it by the corresponding value of a sine wave and of a cosine wave, each at some harmonic frequency of the test waveform. The inner `for()` loop computes these products, and also a running total of each (`sinsum` and `cossum`). After each completion of the inner `for()` loop, we use the Pythagorean Theorem to combine the sine- and cosine-sums so that we get a complete summation (`polarsum`) for that harmonic, saving each one in an array `polarsum[]`, with `polarsum[1]` being the basis for normalizing the values of all others. We then print that summed value. The outer `for()` loop repeats this process for harmonics 1 through 9.
- Our test waveform is generated within its own subroutine, called a *function* in C and C++ alike. Here is where we insert code to generate whatever waveform we wish to analyze. In this particular example, it is a square wave with a peak value of 1. The algorithm for creating this square wave is extremely simple: for x values from 0 to 63 the wave is at $+1$, and for x values from 64 to 127 the wave is at -1. Since the domain of x happens to be 0 to 127 (as called by the `main()` program) this produces one symmetrical cycle of a square wave.

Locating the `f(x)` function within its own section of C++ code allows for easy modification of that function in the future, without modifying the `main()` program. This is generally a good programming practice: to make your code modular so that individual sections of it may be separately edited (and even reside in separate source files!). Doing this makes it easier for teams of programmers to develop projects together, and also makes it easier for code to be re-used in other projects.

7.6.1 DFT of a square wave

When the example code previously shown is compiled and run, the result is the following text output:

```
Harmonic = 1 -- Normalized weight = 1.000000
Harmonic = 2 -- Normalized weight = 0.000000
Harmonic = 3 -- Normalized weight = 0.333601
Harmonic = 4 -- Normalized weight = 0.000000
Harmonic = 5 -- Normalized weight = 0.200483
Harmonic = 6 -- Normalized weight = 0.000000
Harmonic = 7 -- Normalized weight = 0.143548
Harmonic = 8 -- Normalized weight = 0.000000
Harmonic = 9 -- Normalized weight = 0.112009
```

This program assumes the first harmonic's amplitude is the "norm" by which all other harmonics are scaled. Therefore, the first harmonic always shows up as having a normalized weight of 1, with all other harmonic values shown proportionate to that norm.

Fourier theory predicts that a square wave with a 50% duty cycle will only contain odd harmonics (in agreement with our **symmetry rule**), the relative amplitudes of those harmonics diminishing by a factor of $\frac{1}{n}$ where n is the harmonic number. Therefore, if the first harmonic is normalized to an amplitude of 1, then the third harmonic will have an amplitude of $\frac{1}{3}$, the fifth harmonic an amplitude of $\frac{1}{5}$, etc.:

$$v_{square} = \frac{4}{\pi} V_m \left(\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \frac{1}{7} \sin 7\omega t + \cdots + \frac{1}{n} \sin n\omega t \right)$$

- 1st harmonic = $\frac{1}{1} = 1$
- 3rd harmonic = $\frac{1}{3} \approx 0.3333$
- 5th harmonic = $\frac{1}{5} = 0.2000$
- 7th harmonic = $\frac{1}{7} \approx 0.1429$
- 9th harmonic = $\frac{1}{9} \approx 0.1111$

As you can see, the output of our simple DFT algorithm closely approximates these theoretical results.

By modifying just the code within the `f(x)` function we may compute the harmonic content of different wave-shapes. The next several examples will show the modified `f(x)` function code and the resulting output of this DFT algorithm.

7.6.2 DFT of a sine wave

First, we will re-code `f(x)` to generate a simple sine wave. The argument x passed to this function is an integer number starting at zero and incrementing to 127, representing a sequence of samples spanning one period of the fundamental frequency, and so some scaling arithmetic is necessary to convert this domain into a value in radians from 0 to 2π suitable for the `sin()` function:

```
float f(int x)    // Sine wave function
{
  return sin(2 * M_PI * x / 128.0);
}
```

```
Harmonic = 1 -- Normalized weight = 1.000000
Harmonic = 2 -- Normalized weight = 0.000000
Harmonic = 3 -- Normalized weight = 0.000000
Harmonic = 4 -- Normalized weight = 0.000000
Harmonic = 5 -- Normalized weight = 0.000000
Harmonic = 6 -- Normalized weight = 0.000000
Harmonic = 7 -- Normalized weight = 0.000000
Harmonic = 8 -- Normalized weight = 0.000000
Harmonic = 9 -- Normalized weight = 0.000000
```

Not surprisingly, the result is a strong first harmonic and no other harmonics. Also, we get the same results if we replace the sine function with a cosine function in `f(x)`: in either case, a plain sinusoid only has one harmonic component, and that is the first harmonic.

7.6.3 DFT of a delta function

As another test of our DFT algorithm, we will re-code `f(x)` to output a *delta function*, which is nothing more than the briefest of impulses. A delta function consists of a "spike"[14] at time zero followed (and preceded) by values of zero:

```
float f(int x)  // Delta impulse function
{
  if (x == 0)
    return 1.0;

  else
    return 0.0;
}
```

```
Harmonic = 1 -- Normalized weight = 1.000000
Harmonic = 2 -- Normalized weight = 1.000000
Harmonic = 3 -- Normalized weight = 1.000000
Harmonic = 4 -- Normalized weight = 1.000000
Harmonic = 5 -- Normalized weight = 1.000000
Harmonic = 6 -- Normalized weight = 1.000000
Harmonic = 7 -- Normalized weight = 1.000000
Harmonic = 8 -- Normalized weight = 1.000000
Harmonic = 9 -- Normalized weight = 1.000000
```

The result is *all* harmonics at equal strength, which is what the Fourier transform predicts for a delta function: a constant-valued function in the frequency domain. In other words, an infinitesimally brief impulse is equivalent to a superposition of *all* frequencies.

This is a good example of our **steepness rule** in action: a delta function consists of nothing but steepness, being a "spike" up and down over the briefest possible time interval. As such, it contains all frequencies, which of course includes the nine harmonic frequencies shown.

If we consider carefully how the DFT algorithm works, it becomes evident why this must be so, and precisely how every frequency's value must have the same normalized value. The very first sample (`sample = 0`) is the only one where the delta function is not zero, and therefore this will be the only sample where any of the sums tallied in the program accumulate any value. Furthermore, the only sums accumulating value during this sample must be the *cosine* sums because the sine function is zero at an angle of zero, while cosine is one at an angle of zero. Therefore, every cosine function multiplied by the delta impulse function will increment its sum by one. This must include every cosine *of every conceivable frequency* and not just the select harmonics tested by our DFT algorithm. Therefore, based on the criteria of the DFT algorithm, a delta function must contain *all* cosine terms, of *every* frequency.

[14] A true *Dirac delta function* actually consists of an infinite-magnitude spike with zero width, but having an enclosed area equal to unity. We cannot emulate that in procedural code, but we may approximate it!

In practice there is no such thing as a real delta impulse function. A function consisting of a pulse of infinitesimal width defies physical implementation, but nevertheless is useful as a theoretical tool, and serves as a limit for very brief (real) pulses. The practical lesson to learn here is that the spectra of real pulse signals approaches uniformity as the width of the pulse approaches zero – i.e. the briefer the pulse duration, the wider the spread of constituent frequencies. This means any circuitry tasked with amplifying, attenuating, or otherwise processing this pulse signal must contend with a broad span of frequencies, and failure to properly process *all* of the frequencies within that pulse signal invariably corrupts the pulse in some way.

7.6.4 DFT of two sine waves

Next, we will try modifying `f(x)` to generate a superposition of two sine waves, one at 5× our assumed fundamental frequency, and another at 8× the fundamental:

```
float f(int x)  // Dual sine waves
{
  return sin(5 * 2 * M_PI * x / 128.0)
  + sin(8 * 2 * M_PI * x / 128.0);
}
```

From this we would expect a harmonic spectrum consisting of a 5th harmonic and 8th harmonic, and nothing else. What we obtain looks strange at first, though:

```
Harmonic = 1 -- Normalized weight = 1.000000
Harmonic = 2 -- Normalized weight = 3.847159
Harmonic = 3 -- Normalized weight = 4.564931
Harmonic = 4 -- Normalized weight = 5.773269
Harmonic = 5 -- Normalized weight = 159252960.000000
Harmonic = 6 -- Normalized weight = 4.397245
Harmonic = 7 -- Normalized weight = 2.756398
Harmonic = 8 -- Normalized weight = 159252960.000000
Harmonic = 9 -- Normalized weight = 1.914945
```

The amplitudes of the 5th and 8th harmonics are enormous, while the others are meager by comparison. Remember, though, that our DFT algorithm *normalizes* all harmonic amplitudes to that of the first harmonic, which in this particular case should be virtually nonexistent. Therefore, the first harmonic registers with a weight of 1, the 5th and 8th with very large weights, and the others about as small as the first harmonic (in comparison with the 5th and 8th). So, even with the crude nature of this algorithm, we get a spectral response that makes sense for the test waveform.

This is a good example of our **superposition rule**, where the spectrum of two superimposed waves is the superposition of those waves' spectra. The 5th harmonic wave consisted of a single peak in its "spectrum" as did the 8th harmonic wave. When these two waves were added in their time domains, the result is a spectrum consisting of those two frequency peaks, no more and no less.

7.6.5 DFT of an amplitude-modulated sine wave

Next, we will re-code `f(x)` to generate an *amplitude-modulated* waveform: the product of a sine wave at 2× the assumed fundamental and another sine wave at 5× the fundamental.

```
float f(int x)  // Mixed sine waves (AM)
{
  return sin(2 * 2 * M_PI * x / 128.0)
  * sin(5 * 2 * M_PI * x / 128.0);
}
```

Modulation theory predicts that "mixing" two sinusoids in this manner will result in two completely new frequencies: one being the sum of the two mixed frequencies, and the other being the difference of the two mixed frequencies. So, for one sine wave oscillating at 2ω and another at 5ω, we would expect one sinusoid at $(5+2)\omega$ and another at $(5-2)\omega$.

```
Harmonic = 1 -- Normalized weight = 1.000000
Harmonic = 2 -- Normalized weight = 0.077597
Harmonic = 3 -- Normalized weight = 17697600.000000
Harmonic = 4 -- Normalized weight = 0.180189
Harmonic = 5 -- Normalized weight = 0.128424
Harmonic = 6 -- Normalized weight = 0.152171
Harmonic = 7 -- Normalized weight = 17697598.000000
Harmonic = 8 -- Normalized weight = 0.150321
Harmonic = 9 -- Normalized weight = 0.557960
```

True to form, the result is a pair of harmonics in the spectrum, a 3rd harmonic and a 7th harmonic.

This is an excellent example of our **non-linear systems rule**: when signals pass through non-linear systems, new frequencies arise. Multiplication of two independent signals is definitely nonlinear, as doubling both signals' amplitudes does *not* result in a doubled output amplitude. What came into this system was a 2nd and 5th harmonic, but what left was a 3rd and 7th harmonic.

7.6.6 DFT of a full-rectified sine wave

Next, we will re-code `f(x)` to generate the first half (i.e. positive half) of a sine wave. This is all we need to simulate a full-wave rectified sinusoid since all other half-periods of that wave will identical to the first. To represent this in code, we just take the same line used for the sine wave and eliminate the `2` multiplier. In other words, instead of calculating $\sin\left(\frac{2\pi x}{128}\right)$ we compute $\sin\left(\frac{\pi x}{128}\right)$:

```
float f(int x)  // Full-rectified sine wave
{
  return sin(M_PI * x / 128.0);
}
```

The result is shown here:

```
Harmonic = 1 -- Normalized weight = 1.000000
Harmonic = 2 -- Normalized weight = 0.200121
Harmonic = 3 -- Normalized weight = 0.085852
Harmonic = 4 -- Normalized weight = 0.047763
Harmonic = 5 -- Normalized weight = 0.030450
Harmonic = 6 -- Normalized weight = 0.021127
Harmonic = 7 -- Normalized weight = 0.015534
Harmonic = 8 -- Normalized weight = 0.011915
Harmonic = 9 -- Normalized weight = 0.009439
```

Fourier theory predicts the relative amplitudes of each harmonic for a full-rectified sine wave diminish by a factor of $\frac{1}{4n^2-1}$ where n is the harmonic number. Therefore, if the first harmonic has an amplitude of $\frac{1}{3}$, then the second harmonic will have an amplitude of $\frac{1}{15}$, the third harmonic an amplitude of $\frac{1}{35}$, etc. If we normalize all the amplitudes to that of the first harmonic, the relative amplitudes will be as follows:

- 1st harmonic $= \frac{3}{3} = 1$
- 2nd harmonic $= \frac{3}{15} = \frac{1}{5} = 0.200$
- 3rd harmonic $= \frac{3}{35} \approx 0.0857$
- 4th harmonic $= \frac{3}{63} = \frac{1}{21} \approx 0.0476$
- 5th harmonic $= \frac{3}{99} = \frac{1}{33} \approx 0.0303$

As you can see, the output of our simple DFT algorithm closely approximates these theoretical results.

This is a good example of our **symmetry rule**. A rectified sine wave does not have the same shape when inverted, and so we know it must contain even-numbered harmonics. Contrast this against symmetrical waveforms such as the square wave from the original code example, generating a spectrum consisting only of odd-numbered harmonics.

7.7 Spectrum analyzer in C++

This program builds on the foundation of the Discrete Fourier Transform (DFT) from the previous section, but instead of displaying only the normalized harmonic amplitudes this program outputs a comma-separated value (CSV) file that may be plotted using any spreadsheet application (e.g. Microsoft Excel) or mathematical visualizing application (e.g. `gnuplot`).

I happened to use `gnuplot` to generate the spectra. My gnuplot script is as follows, saved to a file named `script.txt`:

```
set datafile separator ","
set xrange [0:10.0]
set style line 1 lw 2 lc rgb "red"
plot 'data.csv' using 1:2 with lines ls 1
```

All C++ programs were compiled using `g++` and run with text output redirected to a file named `data.csv` using the following command-line instructions:

```
g++ main.cpp ; ./a.out > data.csv
```

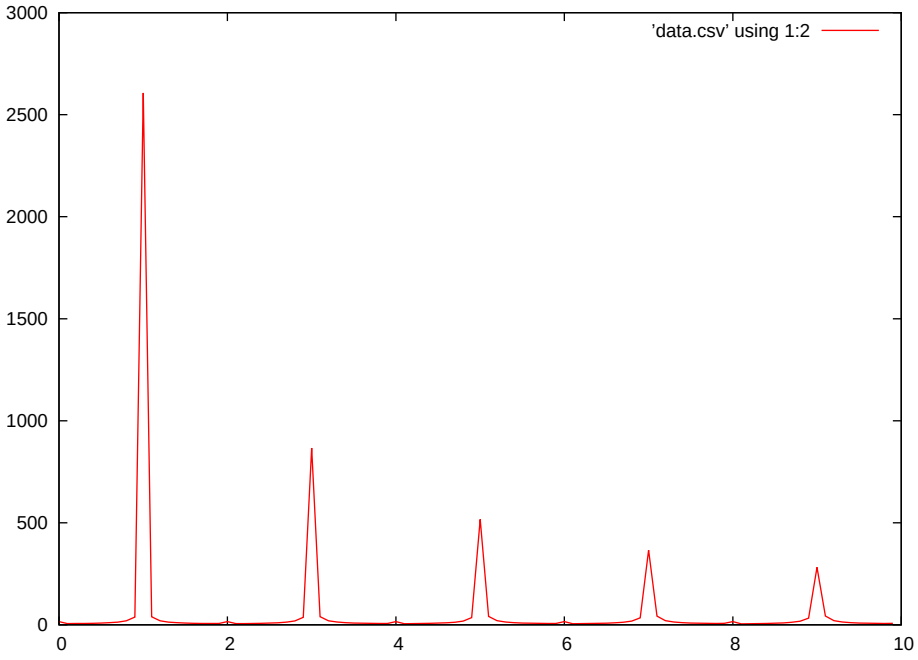
Then, after the comma-separated value file was populated with data from the C++ program's execution, I run `gnuplot` using the following command:

```
gnuplot -p script.txt
```

```
#include <iostream>
#include <math.h>
using namespace std;

#define MAX 4096
#define CYCLES 10

float f(int x);

int main(void)
{
  int sample;
  float iharm, sinsum, cossum, polarsum;

  for (iharm = 0.0; iharm < 10.0; iharm = iharm + 0.1)
    {
      sinsum = 0;
      cossum = 0;

      for (sample = 0; sample < MAX; ++sample)
        {
          sinsum = sinsum + (f(sample) * (sin(CYCLES*sample*iharm*2*M_PI/MAX)));
          cossum = cossum + (f(sample) * (cos(CYCLES*sample*iharm*2*M_PI/MAX)));
        }

      polarsum = sqrt(pow(cossum, 2) + pow(sinsum, 2));

      cout << iharm << " , " << polarsum << endl;
    }

  return 0;
}

float f(int x)
{
  // (return value of function to be analyzed here)
}
```

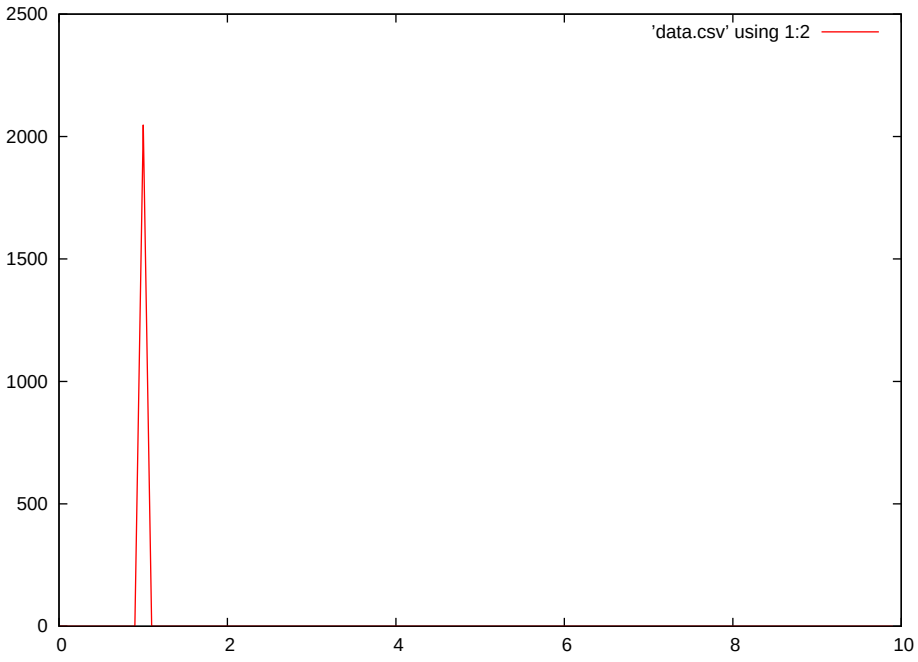
7.7.1 Spectrum of a square wave

```
float f(int x) // Square wave
{
  if ((x % (MAX / CYCLES)) < (0.5 * MAX / CYCLES))
    return 1.0;

  else
    return -1.0;
}
```



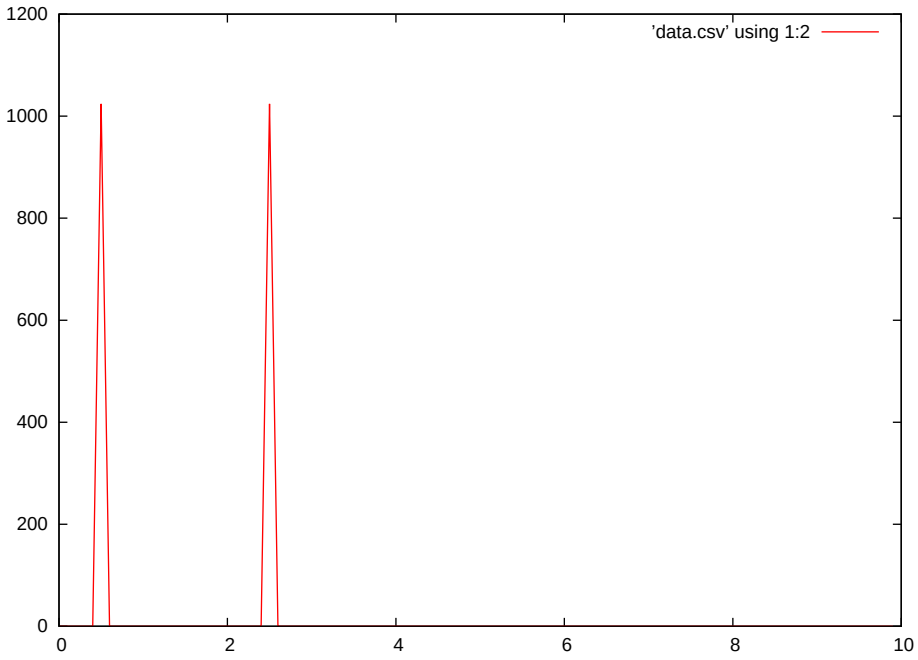
7.7.2 Spectrum of a sine wave

```
float f(int x) // Sine wave
{
  return sin(CYCLES*x*2*M_PI/MAX);
}
```



7.7.3 Spectrum of a sine wave product

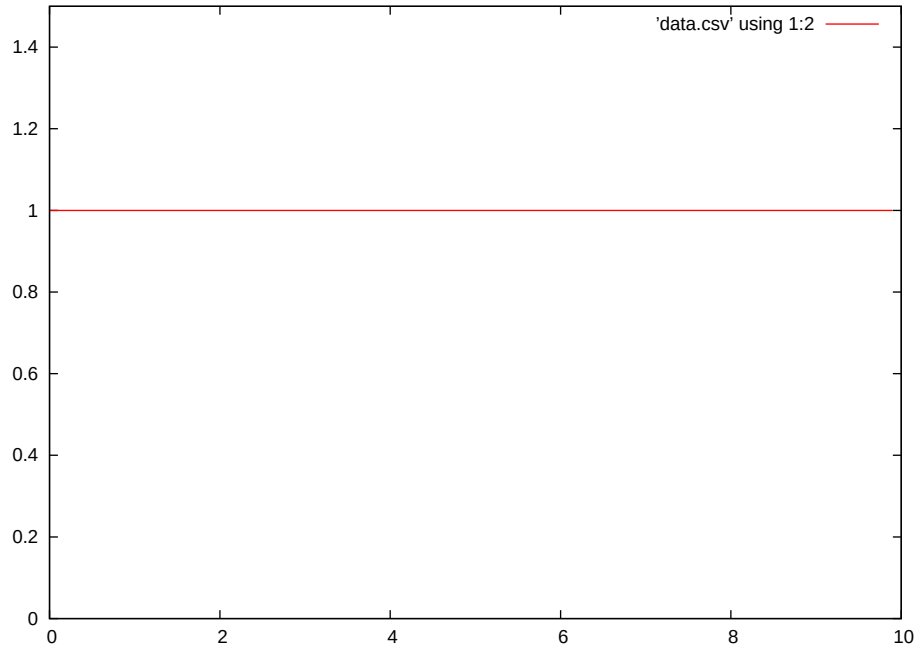
```
float f(int x) // Product of f and 1.5f sine waves
{
  return sin(CYCLES*x*2*M_PI/MAX) * sin(CYCLES*1.5*x*2*M_PI/MAX);
}
```



Note the two peaks at $0.5f$ and $2.5f$: frequencies representing the difference and sum, respectively, of the original sinusoids.

7.7.4 Spectrum of an impulse

```
float f(int x) // Unity impulse function at x = 0
{
  if (x == 0)
    return 1;

  else
    return 0;
}
```



Note how the impulse is equivalent to a spectrum consisting of *all* frequencies. Since the amplitude of the spectrum is much less than in previous examples, I used a different y-axis range in `gnuplot` than in the other simulations:

```
set datafile separator ","
set xrange [0:10.0]
set yrange [0:1.5]
set style line 1 lw 2 lc rgb "red"
plot 'data.csv' using 1:2 with lines ls 1
```

Chapter 8

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component X) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

8.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

8.1.1 Reading outline and reflections

"Reading maketh a full man; conference a ready man; and writing an exact man" – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

8.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

Light

Sinusoidal decomposition (i.e. Fourier's Theorem)

Time domain

Frequency domain

Periodic

Fundamental frequency

Harmonic frequency

Interharmonic frequency

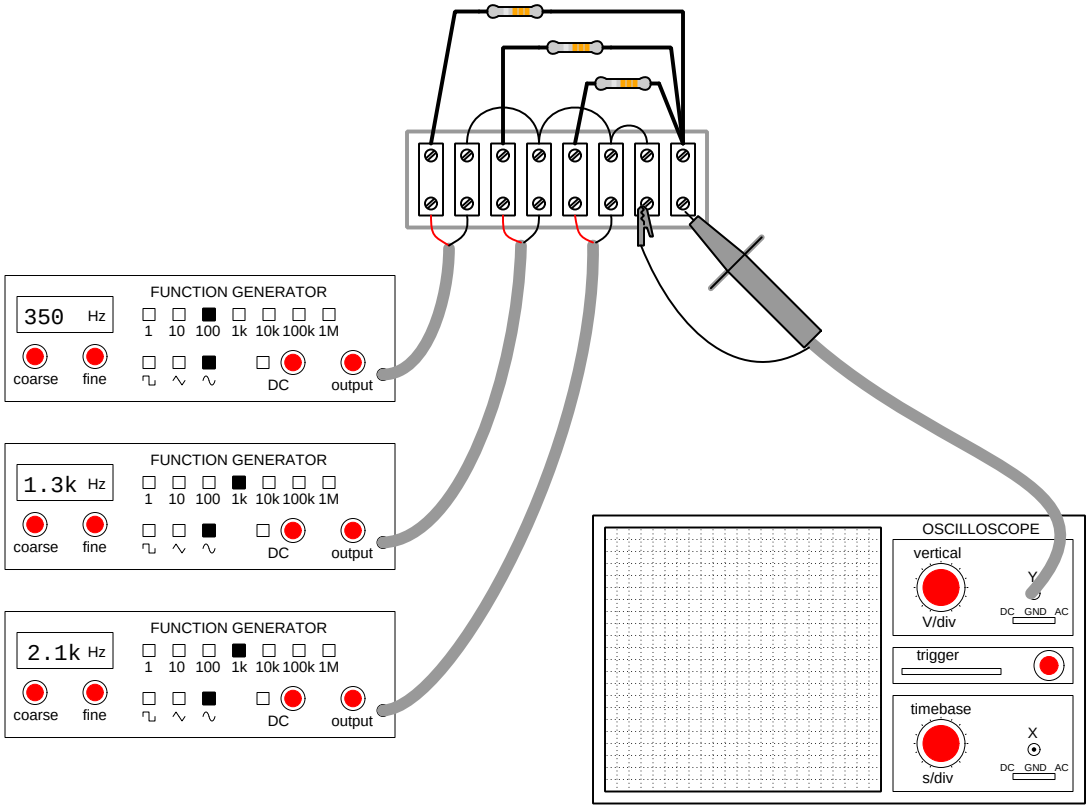
Linearity

Fourier series

Fourier Transform

8.1.3 Combining AC signals

The following circuit combines three AC voltage signals into one, to be measured by an oscilloscope:



First, draw a schematic diagram of this circuit, to make it easier to analyze.

Next, determine whether these three combined AC voltage signals irrevocably affect one another, or if you think it may be possible to separate them into their three original forms.

Last, suppose the combiner circuit contains capacitors and inductors rather than resistors? Would the result be the same? Why or why not?

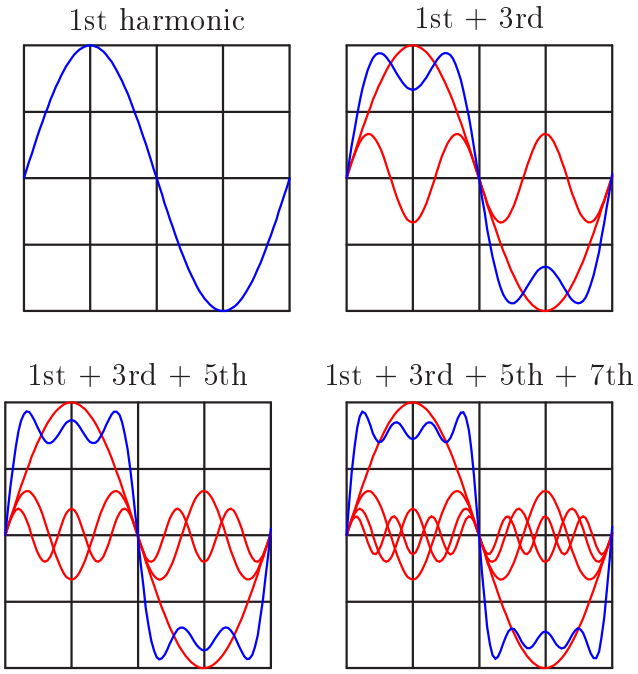
Challenges

- Describe an electrical technique for separating different AC sinusoids that happened to be combined together.
- When electrical "noise" couples to a circuit via parasitic capacitance and/or inductance, can that noise be separated from the true circuit signal, or is the circuit's signal irrevocably tainted by that noise?.

8.1.4 Synthesis of a square wave

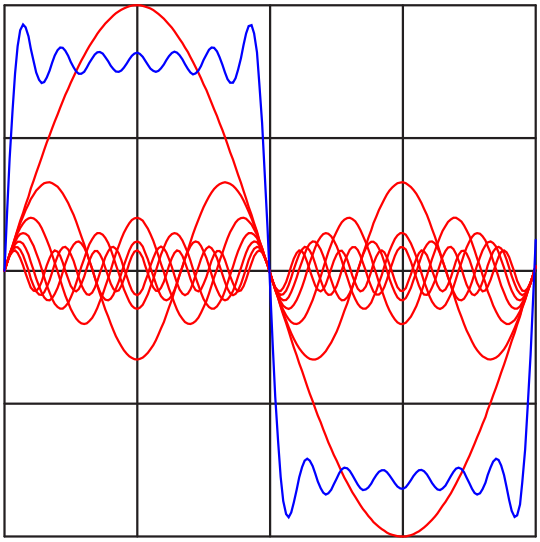
An interesting phenomenon arises when we take the odd-numbered harmonics of a given frequency and add them together at certain diminishing ratios of the fundamental's amplitude. For instance, consider the following harmonic series:

(1 Volt at 100 Hz) + (1/3 Volt at 300 Hz) + (1/5 Volt at 500 Hz) + (1/7 Volt at 700 Hz) + . . .



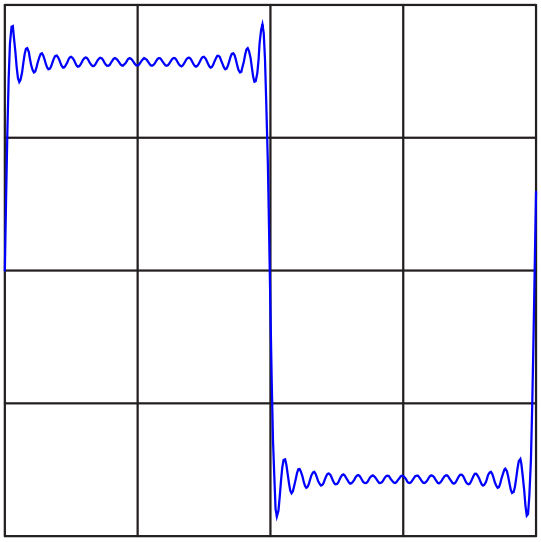
The following graph shows the composite wave of all odd-numbered harmonics up to the 13th together, following the same pattern of diminishing amplitudes:

1st + 3rd + 5th + 7th + 9th + 11th + 13th



If we take this progression even further, you can see that the sum of these harmonics begins to appear more like a square wave:

All odd-numbered harmonics up to the 35th



This mathematical equivalence between a square wave and the weighted sum of all odd-numbered harmonics is very useful in analyzing AC circuits where square-wave signals are present. From the

perspective of AC circuit analysis based on sinusoidal waveforms, how would you describe the way an AC circuit "views" a square wave?

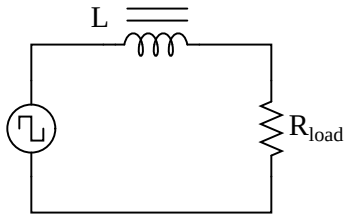
Challenges

- Explain how this equivalence between a square wave and a particular series of sine waves is a practical example of the *Superposition Theorem* at work.

8.1.5 LR circuit energized by a square-wave source

The *Fourier series* is much more than a mathematical abstraction. The mathematical equivalence between any periodic waveform and a series of sinusoidal waveforms can be a powerful analytical tool for the electronic engineer and technician alike.

Explain how knowing the Fourier series for a particular non-sinusoidal waveform simplifies the analysis of an AC circuit. For example, how would our knowledge of a square wave's Fourier series help in the analysis of this circuit?

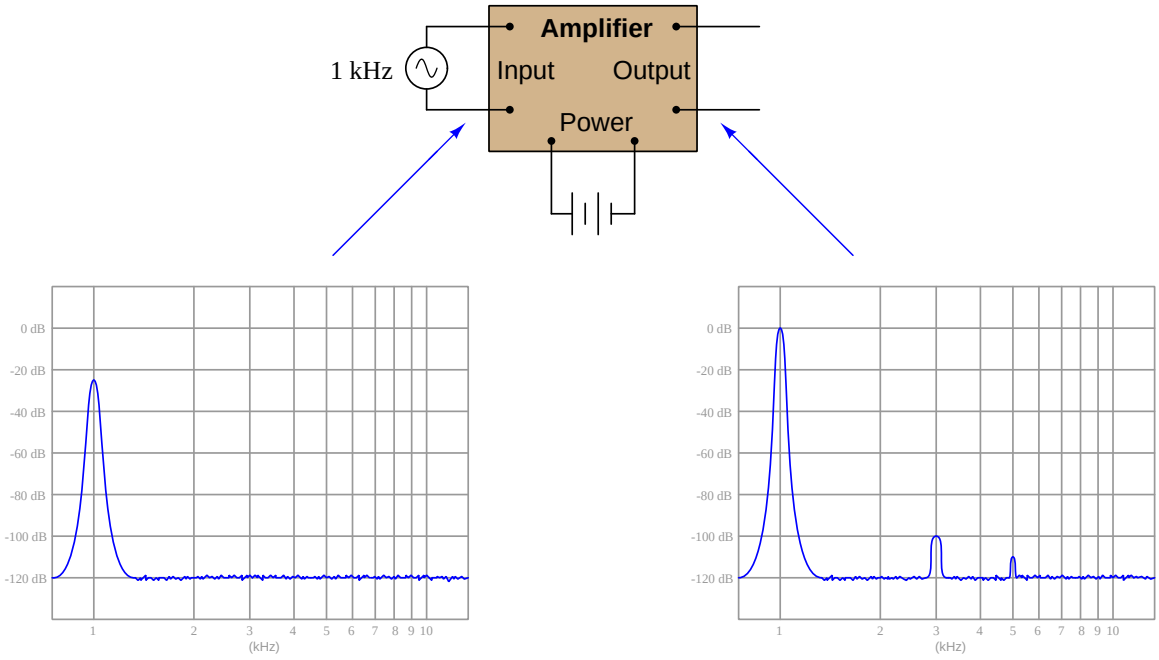


Challenges

- Explain how the Superposition Theorem relates to this problem.

8.1.6 Amplifier test

Suppose an amplifier circuit is connected to a sine-wave signal generator, and a spectrum analyzer used to measure both the input and the output signals of the amplifier:



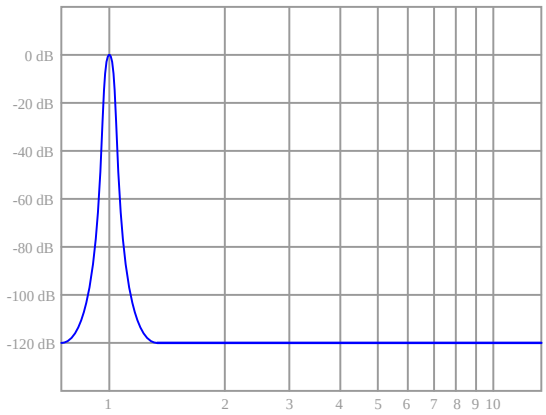
Interpret the two graphical displays and explain why the output signal has more "peaks" than the input. What is this difference telling us about the amplifier's performance?

Challenges

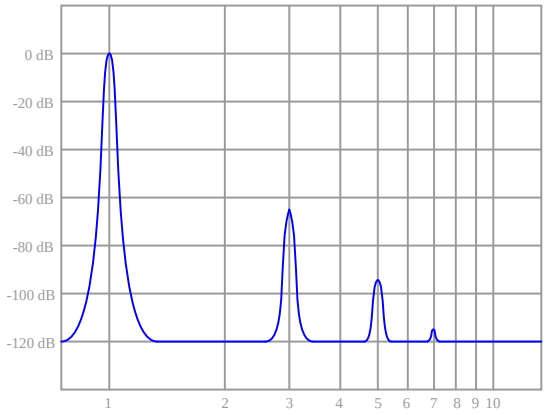
- The spectrum shown in this example is devoid of a *noise floor*. Identify what a "noise floor" is and what it would look like on the spectrum display.
- For those familiar with amplifier circuit design and analysis, identify any specific amplifier problems or component faults that might account for these additional peaks appearing in the output spectrum.

8.1.7 AC line power test

An electronics technician connects the input of a spectrum analyzer to the secondary winding of an AC power transformer, plugged into a power receptacle. She sets the spectrum analyzer to show 60 Hz as the fundamental frequency, expecting to see the following display:



Instead, however, the spectrum analyzer shows more than just a single peak at the fundamental:



Explain what this pattern means, in practical terms. Why is this power system's harmonic signature different from what the technician expected to see?

Challenges

- Suppose it was determined that the electronic load to be powered by this AC line could not tolerate this poor degree of power quality. Explain how we could "condition" this AC power to be suitable for our load.

8.1.8 DC to sunlight

An expression sometimes heard among RF (radio-frequency) engineers and technicians is *"DC to sunlight"* as in the following quote:

> "The model ABC-1234 spectrum analyzer handles signals ranging from DC to sunlight."

Explain what this phrase means.

Challenges

- ???.
- ???.
- ???.

8.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[4]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[5] on an answer key!

[4] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[5] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

8.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number (N_A) = **6.02214076** $\times 10^{23}$ **per mole** (mol^{-1})

Boltzmann's constant (k) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge (e) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant (F) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

Gravitational constant (G) = $6.67430(15) \times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant (R) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant (σ) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**^{4} ($W/m^2 \cdot K^4$)

Speed of light in a vacuum (c) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

8.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

	A	B	C	D
1	Distance traveled	46.9	Kilometers	
2	Time elapsed	1.18	Hours	
3	Average speed	= B1 / B2	km/h	
4				
5				

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[6] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[6] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[7] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[8] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

	A	B
1	x_1	= (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3)
2	x_2	= (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3)
3	a =	9
4	b =	5
5	c =	-2

This example is configured to compute roots[9] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new a, b, and c coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[7] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[8] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[9] Reviewing some algebra here, a *root* is a value for x that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

	A	B	C
1	x_1	= (-B4 + C1) / C2	= sqrt((B4^2) - (4*B3*B5))
2	x_2	= (-B4 - C1) / C2	= 2*B3
3	a =	9	
4	b =	5	
5	c =	-2	

Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[10] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

8.2.3 Harmonics of 60 Hz

What is a *harmonic* frequency? If a particular electronic system (such as an AC power system) has a fundamental frequency of 60 Hz, calculate the frequencies of the following harmonics:

- 1st harmonic =
- 2nd harmonic =
- 3rd harmonic =
- 4th harmonic =
- 5th harmonic =
- 6th harmonic =

Challenges

- In a 60 Hz power system, what might cause harmonics to exist?

8.2.4 Plotting a musical chord

A musical *chord* is a mixture of three or more audible pitches. On an oscilloscope, it would appear to be a very complex waveform, very non-sinusoidal.

Use a graphing calculator, a computer spreadsheet, or some other computer-based plotting tool to graph the sum of the following three frequencies, comprising a *C major* chord:

- 261.63 Hz (middle "C")
- 329.63 Hz ("E")
- 392.00 Hz ("G")

Challenges

- Describe the "envelope" of this combined signal. What shape does the outer-most boundary of the composite wave take, and why do you suppose this happens?

8.2.5 Trigonometric formula

Explain the meaning of the following mathematical formula:

$$f(t) = A_0 + (A_1 \sin \omega t) + (B_1 \cos \omega t) + (A_2 \sin 2\omega t) + (B_2 \cos 2\omega t) + \ldots$$

Challenges

- What sort of waveform might this formula *not* relate to?

8.2.6 Fourier series for a square wave

The Fourier series for a square wave is as follows:

$$v_{square} = \frac{4}{\pi} V_m \left(\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \frac{1}{7} \sin 7\omega t + \cdots + \frac{1}{n} \sin n\omega t \right)$$

Where,
V_m = Peak amplitude of square wave in Volts
ω = Angular velocity of square wave in radians per second
t = Time in seconds
n = An odd integer

Draw the electrical schematic diagram showing multiple sinusoidal AC voltage sources connected together to form an approximate 10 Volt (peak), 200 Hz square wave. Limit yourself to the first four harmonics, labeling each sinusoidal voltage source with its own RMS voltage value and frequency.

Challenges

- To be honest, a four-harmonic equivalent circuit generates a rather poor approximation of a square wave. Explain how to improve it.
- Express the individual voltage source values in Volts peak rather than Volts RMS.

8.2.7 Another Fourier series

Suppose a non-sinusoidal voltage source is represented by the following Fourier series:

$$v(t) = 23.2 + 30\sin(377t) + 15.5\sin\left(1131t + \frac{\pi}{2}\right) + 2.7\sin\left(1508t - \frac{9\pi}{2}\right)$$

Where,

V_m = Peak amplitude of square wave in Volts
ω = Angular velocity of square wave in radians per second
t = Time in seconds

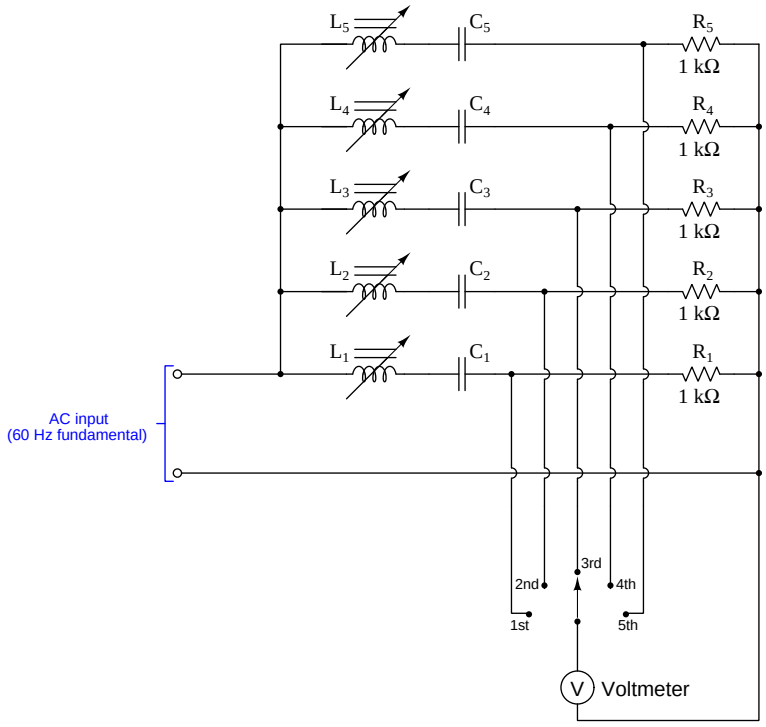
Knowing the Fourier series of this voltage allows us to represent the same voltage source as a set of series-connected voltage sources, each with its own (sinusoidal) frequency. Draw the equivalent schematic in this manner, labeling each voltage source with its RMS voltage value, frequency (in Hz), and phase angle:

Challenges

- Express the individual voltage source values in Volts peak rather than Volts RMS.

8.2.8 AC line harmonic analyzer

Explain how the following power-line harmonic analyzer circuit works:



Harmonic #	$L_{\#}$ value	$C_{\#}$ value
1st	20 to 22 H	0.33 μF
2nd	11 to 12 H	0.15 μF
3rd	5 to 6 H	0.15 μF
4th	1.5 to 2.5 H	0.22 μF
5th	1 to 1.5 H	0.27 μF

If we applied a purely sinusoidal 60 Hz AC input voltage to this analyzer, how would the voltmeter readings compare between the five switch positions?

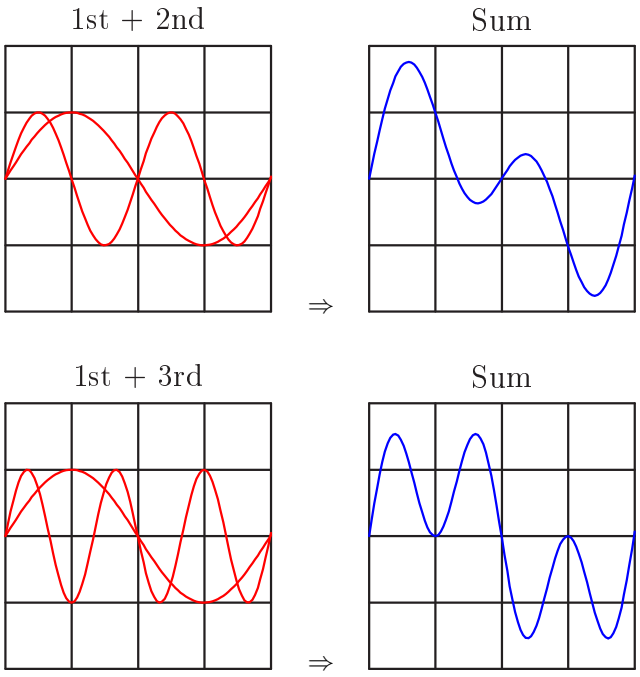
If we applied a 60 Hz *square* AC input voltage to this analyzer, how would the voltmeter readings compare between the five switch positions?

Challenges

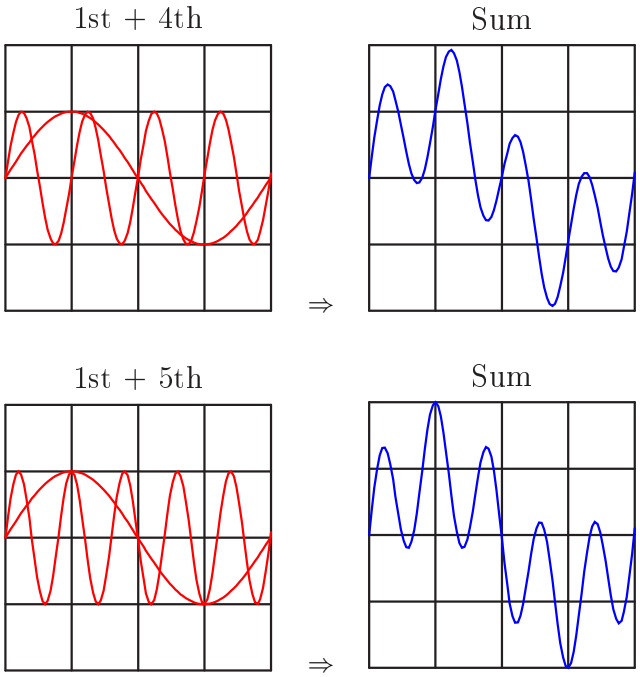
- Calculate the *exact* inductance values necessary for precise tuning of the five LC filters, for the first five harmonics of a 60 Hz waveform.
- The voltmeter in this circuit would not have to be a true-RMS meter. It could simply be an average-responding (RMS-calibrated) voltmeter and it would work the same. Explain why.
- Should the filter networks have high-Q values or low-Q values?

8.2.9 Even versus odd harmonics

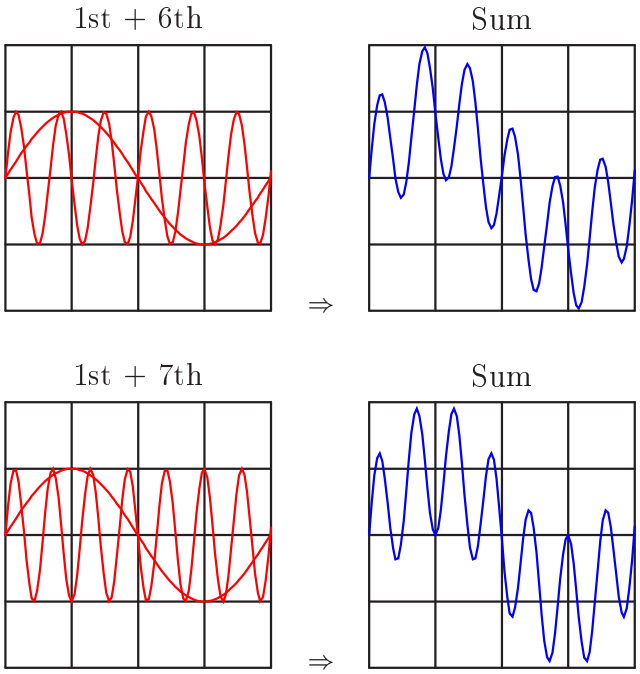
Note the effect of adding the second harmonic of a waveform to the fundamental, and compare that effect with adding the *third* harmonic of a waveform to the fundamental:



Now compare the sums of a fundamental with its fourth harmonic, versus with its fifth harmonic:



And again for the 1st + 6th, versus the 1st + 7th harmonics:



Examine these sets of harmonic sums, and indicate the trend you see with regard to harmonic number and symmetry of the final (Sum) waveforms. Specifically, how does the addition of an *even* harmonic compare to the addition of an *odd* harmonic, in terms of final waveshape?

Challenges

- Identify a practical application of this knowledge.

8.2.10 Multi-harmonic analyzer

Take the following C program and modify it so as to calculate the 1st, 2nd, 3rd, 4th, and 5th harmonics of the sampled waveform (stored in array `x`):

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  double x[100] = {0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                   0.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
                  -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0};
  int sample;
  double cosavg, sinavg;
  cosavg = sinavg = 0.0;

  for (sample = 0 ; sample < 100 ; ++sample)
  {
    cosavg = cosavg + (x[sample] * cos(2 * M_PI * sample / 100));
    sinavg = sinavg + (x[sample] * sin(2 * M_PI * sample / 100));
  }

  printf("DC average value of cosine product = %f\n", cosavg / 100);
  printf("DC average value of sine product = %f\n", sinavg / 100);

  return 0;
}
```

Then, take the results of your edited code and show that the harmonic amplitudes match the amplitude coefficients of the Fourier series for a square wave.

Challenges

- Suppose the waveform was sampled at twice the rate as the one in this example, so that one full cycle spanned 200 samples instead of 100 samples. What would need to change in this program to accommodate the additional data?

8.2.11 Fourier analysis of a triangle wave

Write (or modify) a computer program to analyze a sampled waveform representing a *triangle wave*, and determine its harmonic content.

Challenges

- Does the number of samples in one cycle of the input waveform matter to our analysis?

8.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

8.3.1 Harmonics produced by inductive components

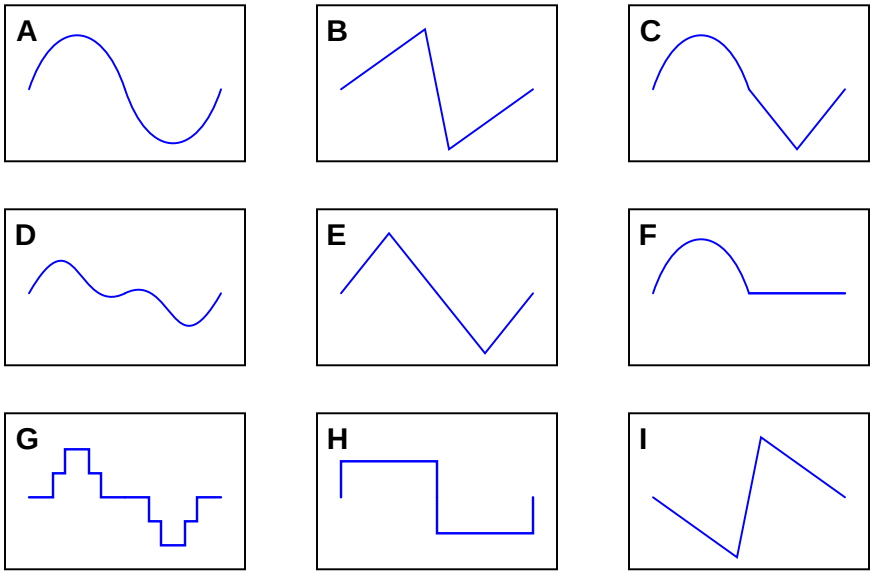
Under certain conditions, harmonics may be produced in AC power systems by inductors and transformers. How is this possible, as these devices are normally considered to be linear?

Challenges

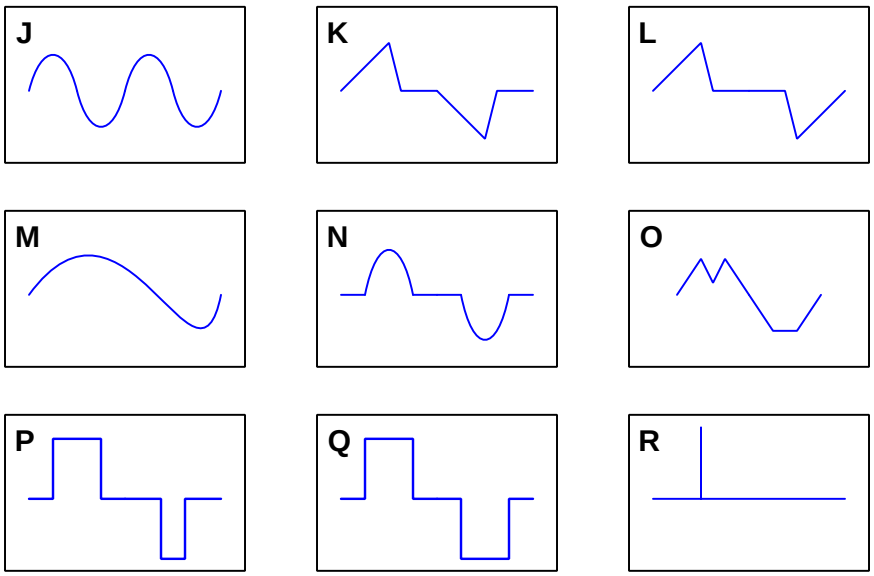
- Does the core material for the inductor or transformer matter?
- Can a given inductor or transformer be operated in such a way as to reduce the production of harmonics? If so, how?

8.3.2 Discerning even/odd harmonics from the time domain

By visual inspection, determine which of the following periodic waveforms displayed in the time domain contain even-numbered harmonics:



Here are some hints to get you started: *waveforms A, E, and G* contain no even-numbered harmonics; *waveforms B, C, and D* contain even-numbered harmonics.



Challenges

- Identify a practical application of being able to discern even/odd harmonics from time-domain plots.
- One of these waveforms contains just *one* harmonic – identify which one.
- One of these waveforms contains *all* frequencies – identify which one.
- Waveforms K and L look very similar, but their harmonic spectra are quite different. Which of these is symmetrical about the centerline and which of these isn't?

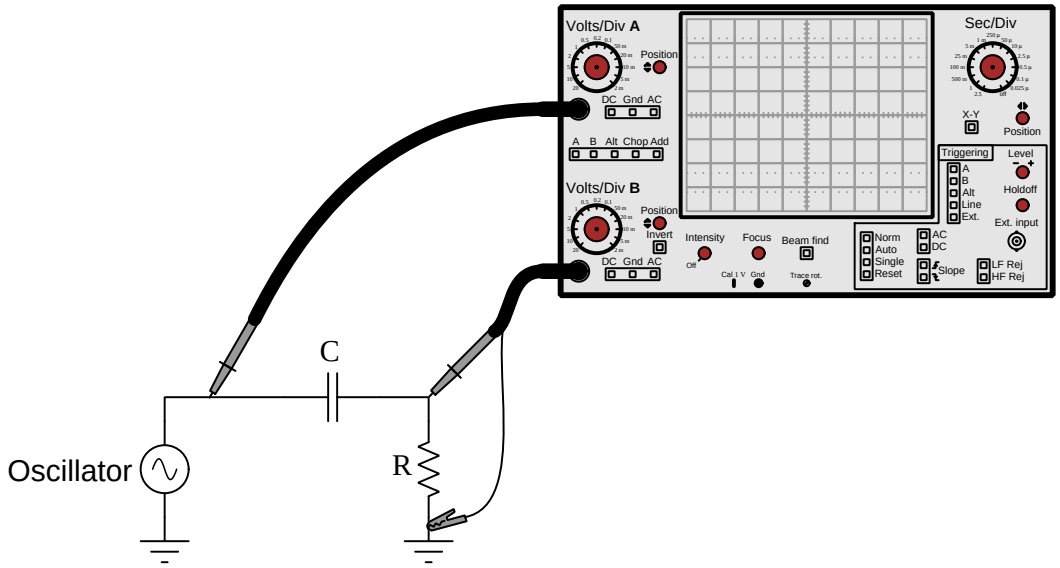
8.3.3 Testing the purity of a sine wave

Suppose a student is developing a sine-wave oscillator circuit and wants to test its output signal to determine whether or not the waveform is acceptably "pure" (i.e. is a true sinusoidal shape).

One way to test the output of this oscillator circuit is by using a spectrum analyzer. Explain how one would interpret the spectrum display given by this instrument to test the purity of the oscillator's waveform.

Assuming the student is on a restricted budget and only has access to an oscilloscope (with no provision for FFT analysis), one could examine the waveform to see that its shape was not grossly distorted, but this would not be a very sensitive test for distortion. The difference between the waveshape of a pure sinusoid and one with just a few percent of harmonic distortion is imperceptible by plain inspection of its time-domain oscillograph.

However, all hope is not lost. The following filter circuit will help reveal whether or not the oscillator's output is a pure sine wave:



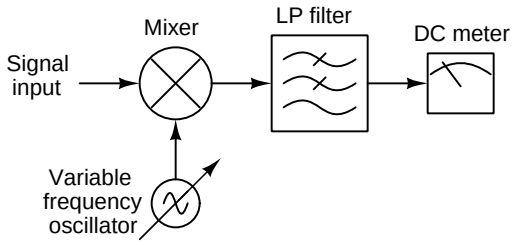
Explain how this oscilloscope-based test will work. What results would you expect if the oscillator output is pure? What results would you expect if its output contains harmonics?

Challenges

- What type of filter network is this?
- Does the cutoff frequency of this filter circuit matter? Why or why not?

8.3.4 Faulty spectrum analyzer design

Suppose someone tried building their own manually-tuned spectrum analyzer, shown in block-diagram form here:



However, they made a mistake. Instead of a low-pass filter, they installed a high-pass filter. What effect would this have on the operation of the analyzer?

Challenges

- What is the proper block-diagram symbol for this incorrect filter type?

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

B.1 First principles of learning

- **Anyone can learn anything** given appropriate time, effort, resources, challenges, encouragement, and expectations. Dedicating time and investing effort are the student's responsibility; providing resources, challenges, and encouragement are the teacher's responsibility; high expectations are a responsibility shared by both student and teacher.
- **Transfer is not automatic.** The human mind has a natural tendency to compartmentalize information, which means the process of taking knowledge learned in one context and applying it to another usually does not come easy and therefore should never be taken for granted.
- **Learning is iterative.** The human mind rarely learns anything perfectly on the first attempt. Anticipate mistakes and plan for multiple tries to achieve full understanding, using the lessons of those mistakes as feedback to guide future attempts.
- **Information is absorbed, but understanding is created.** Facts and procedures may be memorized easily enough by repeated exposure, but the ability to reliably apply principles to novel scenarios only comes through intense personal effort. This effort is fundamentally creative in nature: explaining new concepts in one's own words, running experiments to test understanding, building projects, and teaching others are just a few ways to creatively apply new knowledge. These acts of making knowledge "one's own" need not be perfect in order to be effective, as the value lies in the activity and not necessarily the finished product.
- **Education trumps training.** There is no such thing as an entirely isolated subject, as all fields of knowledge are connected. Training is narrowly-focused and task-oriented. Education is broad-based and principle-oriented. When preparing for a life-long career, education beats training every time.
- **Character matters.** Poor habits are more destructive than deficits of knowledge or skill. This is especially true in collective endeavors, where a team's ability to function depends on trust between its members. Simply put, no one wants an untrustworthy person on their team. An essential component of education then, is character development.
- **People learn to be responsible by bearing responsibility.** An irresponsible person is someone who has never *had* to be responsible for anything that mattered enough to them. Just as anyone can learn anything, anyone can become responsible if the personal cost of irresponsibility becomes high enough.
- **What gets measured, gets done.** Accurate and relevant assessment of learning is key to ensuring all students learn. Therefore, it is imperative to measure what matters.
- **Failure is nothing to fear.** Every human being fails, and fails in multiple ways at multiple times. Eventual success only happens when we don't stop trying.

B.2 Proven strategies for instructors

- Assume every student is capable of learning anything they desire given the proper conditions. Treat them as capable adults by granting real responsibility and avoiding artificial incentives such as merit or demerit points.
- Create a consistent culture of high expectations across the entire program of study. Demonstrate and encourage patience, persistence, and a healthy sense of self-skepticism. Anticipate and de-stigmatize error. Teach respect for the capabilities of others as well as respect for one's own fallibility.
- Replace lecture with "inverted" instruction, where students first encounter new concepts through reading and then spend class time in Socratic dialogue with the instructor exploring those concepts and solving problems individually. There is a world of difference between observing someone solve a problem versus actually solving a problem yourself, and so the point of this form of instruction is to place students in a position where they *cannot* passively observe.
- Require students to read extensively, write about what they learn, and dialogue with you and their peers to sharpen their understanding. Apply Francis Bacon's advice that "reading maketh a full man; conference a ready man; and writing an exact man". These are complementary activities helping students expand their confidence and abilities.
- Use artificial intelligence (AI) to challenge student understanding rather than merely provide information. Find productive ways for AI to critique students' clarity of thought and of expression, for example by employing AI as a Socratic-style interlocutor or as a reviewer of students' journals. Properly applied, AI has the ability to expand student access to critical review well outside the bounds of their instructor's reach.
- Build frequent and rapid feedback into the learning process so that students know at all times how well they are learning, to identify problems early and fix them before they grow. Model the intellectual habit of self-assessing and self-correcting your own understanding (i.e. a cognitive *feedback loop*), encouraging students to do the same.
- Use "mastery" as the standard for every assessment, which means the exam or experiment or project must be done with 100% competence in order to pass. Provide students with multiple opportunity for re-tries (different versions of the assessment every time).
- Require students to devise their own hypotheses and procedures on all experiments, so that the process is truly a scientific one. Have students assess their proposed experimental procedures for risk and devise mitigations for those risks. Let nothing be pre-designed about students' experiments other than a stated task (i.e. what principle the experiment shall test) at the start and a set of demonstrable knowledge and skill objectives at the end.
- Have students build as much of their lab equipment as possible: building power sources, building test assemblies[1], and building complete working systems (no kits!). In order to provide

[1] In the program I teach, every student builds their own "Development Board" consisting of a metal chassis with DIN rail, terminal blocks, and an AC-DC power supply of their own making which functions as a portable lab environment they can use at school as well as take home.

this same "ground-up" experience for every new student, this means either previous students take their creations with them, or the systems get disassembled in preparation for the new students, or the systems grow and evolve with each new student group.

- Incorporate external accountability for you and for your students, continuously improving the curriculum and your instructional methods based on proven results. Have students regularly network with active professionals through participation in advisory committee meetings, service projects, tours, jobshadows, internships, etc. Practical suggestions include requiring students to design and build projects for external clients (e.g. community groups, businesses, different departments within the institution), and also requiring students attend all technical advisory committee meetings and dialogue with the industry representatives attending.

- Repeatedly explore difficult-to-learn concepts across multiple courses, so that students have multiple opportunities to build their understanding.

- Relate all new concepts, whenever possible, to previous concepts and to relevant physical laws. Challenge each and every student, every day, to *reason* from concept to concept and to explain the logical connections between. Challenge students to verify their conclusions by multiple approaches (e.g. double-checking their work using different methods). Ask *"Why?"* often.

- Maintain detailed records on each student's performance and share these records privately with them. These records should include academic performance as well as professionally relevant behavioral tendencies.

- Address problems while they are small, before they grow larger. This is equally true when helping students overcome confusion as it is when helping students build professional habits.

- Build rigorous quality control into the curriculum to ensure every student masters every important concept, and that the mastery is retained over time. This includes (1) review questions added to every exam to re-assess knowledge taught in previous terms, (2) cumulative exams at the end of every term to re-assess all important concepts back to the very beginning of the program, and (3) review assessments in practical (hands-on) coursework to ensure critically-important skills were indeed taught and are still retained. What you will find by doing this is that it actually boosts retention of students by ensuring that important knowledge gets taught and is retained over long spans of time. In the absence of such quality control, student learning and retention tends to be spotty and this contributes to drop-out and failure rates later in their education.

- Finally, *never rush learning.* Education is not a race. Give your students ample time to digest complex ideas, as you continually remind yourself of just how long it took you to achieve mastery! Long-term retention and the consistently correct application of concepts are always the result of *focused effort over long periods of time* which means there are no shortcuts to learning.

B.3 Proven strategies for students

The single most important piece of advice I have for any student of any subject is to take responsibility for your own development in all areas of life including mental development. Expecting others in your life to entirely guide your own development is a recipe for disappointment. This is just as true for students enrolled in formal learning institutions as it is for auto-didacts pursuing learning entirely on their own. Learning to think in new ways is key to being able to gainfully use information, to make informed decisions about your life, and to best serve those you care about. With this in mind, I offer the following advice to students:

- **Approach all learning as valuable**. No matter what course you take, no matter who you learn from, no matter the subject, there is something useful in every learning experience. If you don't see the value of every new experience, you are not looking closely enough!
- **Continually challenge yourself**. Let other people take shortcuts and find easy answers to easy problems. The purpose of education is to stretch your mind, in order to shape it into a more powerful tool. This doesn't come by taking the path of least resistance. An excellent analogy for an empowering education is productive physical exercise: becoming stronger, more flexible, and more persistent only comes through intense personal effort.
- **Master the use of language**. This includes reading extensively, writing every day, listening closely, and speaking articulately. To a great extent language channels and empowers thought, so the better you are at wielding language the better you will be at grasping abstract concepts and articulating them not only for your benefit but for others as well.
- **Do not limit yourself to the resources given to you**. Read books that are not on the reading list. Run experiments that aren't assigned to you. Form study groups outside of class. Take an entrepreneurial approach to your own education, as though it were a business you were building for your future benefit.
- **Express and share what you learn**. Take every opportunity to teach what you have learned to others, as this will not only help them but will also strengthen your own understanding[2].
- Realize that **no one can give you understanding**, just as no one can give you physical fitness. These both must be *built*.
- **Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable**. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[3] effort, and never give up! That concepts don't immediately come to you is not a sign of something wrong, but rather of something right: that you have found a worthy challenge!

[2] On a personal note, I was surprised to learn just how much my own understanding of electronics and related subjects was strengthened by becoming a teacher. When you are tasked every day with helping other people grasp complex topics, it catalyzes your own learning by giving you powerful incentives to study, to articulate your thoughts, and to reflect deeply on the process of learning.

[3] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

B.4 Design of these learning modules

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits. Every effort has been made to embed the following instructional and assessment philosophies within:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[4] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[5] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[6] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[4] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[5] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[6] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Appendix E

References

Fleming, John Ambrose, *The Principles of Electric Wave Telegraphy and Telephony*, Second Edition, Longmans, Green, and Co., London, 1910.

Halpin, Mark, editor, *Tutorial on Harmonics Modeling and Simulation*, document TP-125-0, IEEE Power Engineering Society, Piscataway, NJ, 1998.

Kaplan, Wilfred, *Advanced Mathematics for Engineers*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

Lathi, B.P., *Signals, Systems and Communication*, John Wiley & Sons, Inc., New York, 1965.

Oliver, B.M., "Square Wave & Pulse Testing of Linear Systems", Application Note 17, Hewlett-Packard, Palo Alto, 1 April 1966.

Osgood, Brad, "Lecture Notes for EE 261 (The Fourier Transform and its Applications)".

Rutter, H. Thornton, *Modern Motors – Their Construction, Management and Control*, Volume I, Virtue & Co., London, 1922.

Smith, Steven W., *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, San Diego, 1997.

"Spectrum Analysis – Spectrum Analyzer Basics", Application Note 150, Hewlett-Packard, Palo Alto, CA, April 1974.

Steinmetz, Charles Proteus, *Theory and Calculation of Electric Circuits*, First Edition, Sixth Impression, McGraw-Hill Book Company, Inc., New York, 1917.

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

1-2 October 2024 – added more instructor notes. Also eliminated one of the Quantitative Reasoning questions ("Resistor powered by a square-wave source") and added more instructor notes to the "Another Fourier series" Quantitative Reasoning question.

15 September 2024 – divided the Introduction chapter into sections, one with recommendations for students, one with a listing of challenging concepts, and one with recommendations for instructors.

11 March 2024 – edited image_5852 to show X-Y mode being used on the oscilloscope, and also added more explanatory text on this oscilloscope mode.

3-4 March 2024 – edits to some instructor notes, as well as corrected a couple of mis-statements about the harmonic amplitudes of a pulse waveform (not being simply $\frac{1}{n}$).

27 February 2024 – added more questions to the Introduction chapter.

23 February 2024 – typo correction courtesy of David Mitchell, where I had one of the harmonics for a triangle wave listed as the sine function rather than cosine as it should have been.

5 October 2023 – minor edits to the Full Tutorial text.

27-28 July 2023 – fixed a typographical error in the Full Tutorial section on synthesizing a triangle wave, where the final Fourier series had incorrect coefficients on the third- and fifth-harmonic terms.

16 July 2023 – converted the Tutorial chapter into a Full Tutorial, and added a Simplified Tutorial chapter.

26 May 2023 – modified the "Resistor powered by a square-wave source" Quantitative Reasoning

question to include a coding challenge.

24-26 February 2023 – added magnitude coefficients to the Fourier Series shown for square, triangle, and sawtooth waves in the "Fourier series for common waveforms" section of the Technical References chapter. Also, added another section to the Tutorial showing the synthesis of a pulse wave from cosine waves.

18 February 2023 – added another section to the Tutorial showing the synthesis of a sawtooth wave from sine waves.

16 February 2023 – added some clarifying text in the Tutorial, and added two new sections to it showing the synthesis of square and triangle waves from odd-harmonic sine and cosine waves, respectively.

28-29 November 2022 – placed questions at the top of the itemized list in the Introduction chapter prompting students to devise experiments related to the tutorial content.

12 July 2022 – minor additions to the Case Tutorial section on sine wave versus non-sine wave signal sources when experimenting with AC networks.

24 May 2022 – minor additions to the Tutorial, discussing harmonics. Also added some minor details to the Technical Reference section on common Fourier series to better describe the meaning of the coefficients.

3 March 2022 – added more Challenge questions.

3-4 January 2022 – added new section on digital Fourier transforms, as well as new content explaining Fourier series in more detail. Also added some questions.

2 January 2022 – added new content on manually analyzing a square wave.

31 December 2021 – added new sections on analog Fourier analysis techniques.

22 December 2021 – divided Tutorial into sections.

18 November 2021 – placed Discrete Fourier Transform and Spectrum Analyzer C++ programming references into their own files, sharing them with the Elementary Filters module.

3 November 2021 – explicitly stated Fourier's Theorem.

29 September 2021 – added SDR screenshot showing lightning strikes producing broad-spectrum radio interference.

3 June 2021 – added Case Tutorial section discussing the use of triangle-wave AC sources versus sine-wave AC sources to power simple RLC networks.

8 May 2021 – commented out or deleted empty chapters.

26 April 2021 – minor additions to the "DFT of a square wave" subsection of the "Programming References" chapter, showing tabulated values for a square wave's harmonic amplitudes to compare against the DFT algorithm's printed values.

16 April 2021 – added a Case Tutorial chapter.

18 March 2021 – corrected one instance of "volts" that should have been capitalized "Volts".

8 March 2021 – added "DC to sunlight" Conceptual question.

2 March 2021 – minor edit to the "Discerning even/odd harmonics from the time domain" Diagnostic question.

6 January 2021 – added an impulse function simulation to the "Spectrum Analyzer in C++" Programming References section.

6 October 2020 – converted the Simplified Tutorial chapter into (just) a Tutorial chapter, and made some edits to the text as well.

29 September 2020 – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading "inverted" teaching sessions. Also made some minor edits to the Simplified Tutorial.

7 August 2020 – added a Programming Reference section using C++ to plot the continuous spectrum of any arbitrary function.

30 April 2020 – added a Diagnostic Reasoning question on discerning imperfect sine waves using a high-pass filter network and an oscilloscope.

29 April 2020 – added more common Fourier series to that Technical Reference section, and placed each one in its own subsection.

28 April 2020 – added screenshot of spectrum analyzer display showing noise floor, and edited the "Ideal" spectrum display shown in the Tutorial.

27 April 2020 – added another Technical Reference section, this one showing Fourier series for some common functions.

20 April 2020 – added a Programming Reference section using C++ to plot two sinusoidal waveforms with their own phase shifts.

15 April 2020 – minor edits to instructor notes, and edited two of the waveform images in the "Discerning even/odd harmonics from the time domain" Diagnostic Reasoning problem. Also added more content to the Tutorial talking about noise floors.

3 February 2020 – minor edit to one of the Foundational Concept titles.

29 January 2020 – added Foundational Concepts to the list in the Conceptual Reasoning section.

5 January 2020 – added bullet-list of relevant programming principles to the Programming References section.

4 January 2020 – added Programming References chapter, with section on plotting simple sine waves to the console.

2 January 2020 – removed from from C++ code execution output, to clearly distinguish it from the source code listing which is still framed.

15 December 2019 – added reference to the IEEE *Tutorial on Harmonics Modeling and Simulation*, and also the term *interharmonic* within the Simplified Tutorial.

1 December 2019 – added more explanatory text on the Discrete Fourier Transform algorithm (coded in C++).

29 November 2019 – added Technical Reference on Discrete Fourier Transform algorithm (coded in C++).

27 November 2019 – continued adding questions.

27 November 2019 – continued writing Simplified Tutorial, added Historical References on vibrating-reed frequency meters and "wave screens", and added multiple questions, and added source code in the Technical Reference chapter for a Discrete Fourier Transform (DFT) algorithm.

25 November 2019 – continued writing Simplified Tutorial.

24 November 2019 – began writing Tutorial.

23 November 2019 – added a few Questions.

18 November 2019 – rough document first created.

Index