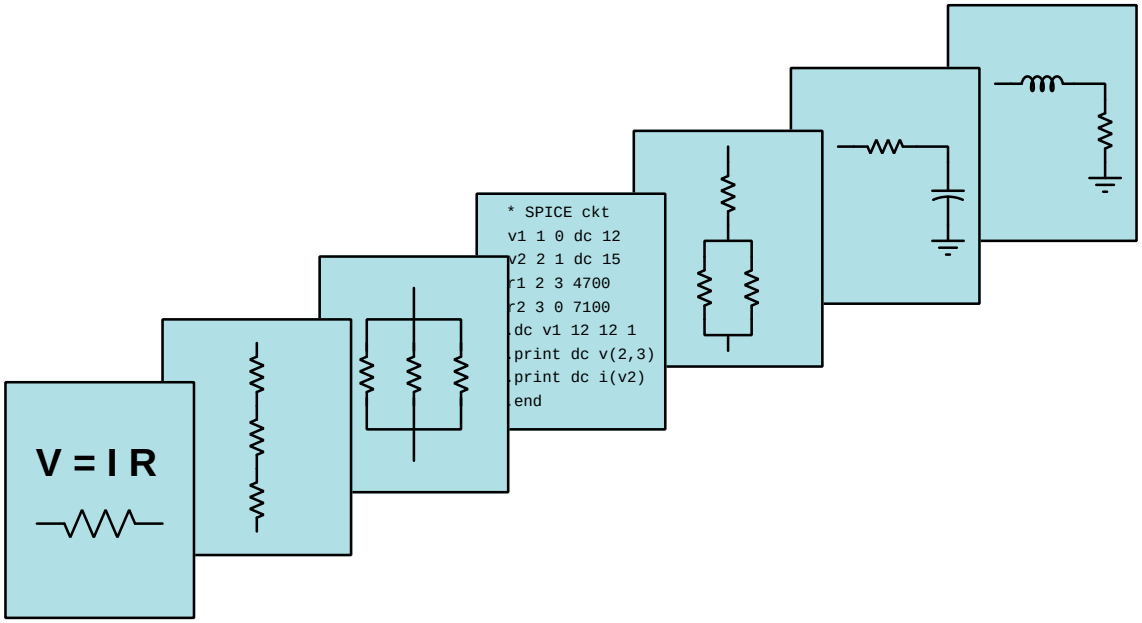
Modular Electronics Learning (ModEL) project



AC Quantities and Measurements

Last update = 25 August 2024

Contents

Chapter 1

Introduction

1.1 Recommendations for students

Alternating current, or *AC*, is any current or voltage that switches direction over time, usually following a sinusoidal-shaped trajectory. A challenge with AC is how to quantify it, because its amplitude constantly varies. However, if that variation is *periodic*, we may determine how often it reverses direction, the time between reversals, the peak values it reaches, etc.

Most of the electrical power in the world is conveyed in AC form rather than DC. This is historically due to two factors: AC electric motors are simpler and therefore inherently more reliable than DC electric motors; and, AC voltage and current may be proportionately stepped up or down by a device called a *transformer* for the purpose of high-efficiency transport over long distances. For many years the transformation of DC voltage and current could not be performed as efficiently as it is for AC, which made AC the clear choice for electric power systems. Modern technology has closed that gap, but electric power grids are and will likely remain AC for a very long time.

Important concepts related to AC include **energy**, **voltage**, **sources** versus **loads**, **polarity**, **oscillations**, **generators** and **motors**, **Faraday's Law of Electromagnetic Induction**, **transformers**, **primary** versus **secondary** coils, the mathematical **sine** function, **degrees** of angle, **Ohm's Law**, **Joule's Law**, **Kirchhoff's Voltage Law**, **Kirchhoff's Current Law**, **period**, **frequency**, **wavelength**, energy **dissipation**, **RMS** values, **wave-shapes**, electrical **meters**, and **phase**.

AC circuit analysis makes extensive use of trigonometric functions such as *sine* and *cosine*. An active reading strategy to apply in this Tutorial is using a scientific calculator to compute sine values for some of the angle values shown in the graphs and see for yourself how the sine values for those angles match the graphs presented.

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

- How might an experiment be designed and conducted to measure the turns ratio of a transformer? 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 measure the frequency of AC in an electrical power system? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What applications are there for AC?
- What does it mean for an electrical component to be a *source*?
- What does it mean for an electrical component to be a *load*?
- Why is AC used instead of DC for most electrical power systems?
- What does a transformer do, and how does one work?
- How may we specify the amplitude of an AC quantity?
- How may we specify the rate at which an AC quantity alternates?
- How does period relate to frequency?
- When and where do familiar DC circuit laws apply to AC circuits?
- What is *wavelength*, and how is it measured?
- How does wavelength relate to frequency?
- How may we rate the energy-delivery capability of an AC voltage/current in terms comparable with DC?
- Why is the RMS equivalent value of a square wave so much larger than that of a sine wave given the same peak value?
- What does it mean if one AC waveform *leads* or *lags* another?

Readers with an interest in music theory may find the *Derivations and Technical References* section "Musical pitch" helpful in relating those concepts with AC electrical signals.

1.2 Challenging concepts related to alternating current

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

- **RMS versus Peak versus Average** – the very idea of assigning a fixed number for AC voltage or current that (by definition) constantly changes magnitude and direction seems strange. Consequently, there is more than one way to do it. We may assign that value according to the *highest* magnitude reached in a cycle, in which case we call it the *peak* measurement. We may mathematically integrate the waveform over time to figure the mean magnitude, in which case we call it the *average* measurement. Or we may figure out what level of DC (voltage or current) causes the exact same amount of average power to be dissipated by a standard resistive load, in which case we call it the *RMS* measurement. One common mistake here is to think that the relationship between RMS, average, and peak measurements is a matter of fixed ratios. The value "0.707" is memorized by every beginning electronics student as the ratio between RMS and peak, but what is commonly overlooked is that this particular ratio holds true *for perfect sine-waves only!* A wave with a different shape will have a different mathematical relationship between peak and RMS values.
- **Phase shift** – like voltage, phase shift is a relative quantity. A single AC waveform, without another waveform to compare to, cannot have a phase value at all. Only when two waves of the same frequency are compared against one another may we intelligently state a phase shift, or time displacement, between the two. Properly reading phase shifts from an oscillograph can also be challenging for students, and requires reminding that time goes from left to right on an oscillograph: i.e. the wave reaching its peak first (left) compared to the other is leading, while the other is lagging. Another confusing aspect of phase shift is that it may be expressed in multiple ways: e.g. $+90^o$ phase shift is equivalent to -270^o phase shift, since waves repeat their patterns every 360 degrees.

Time spent with an oscilloscope and a sine-wave signal generator is often very helpful to understand the properties of AC waveforms, especially if the oscilloscope provides RMS, peak, and other analytical features to check one's interpretation of the waveform against divisions marked on the display. The "First-time oscilloscope experiment ideas" section of the *Case Tutorial* chapter is a good one to follow for initial oscilloscope experiments.

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 the concepts of period, frequency, and amplitude to oscillographs

  Assessment – Given an oscillograph of an AC voltage or current waveform, measure its period, its frequency, and its amplitude; e.g. pose problems in the form of the "AC voltage oscillographs" Quantitative Reasoning question.

- **Outcome** – Apply the concept of RMS measurement to sine, square, and triangle waveforms

  Assessment – Apply a sine wave from a signal generator to the input of a digital oscilloscope, then predict its RMS value based on peak measurements. Then, confirm the accuracy of that prediction using either the oscilloscope's RMS measurement capability or a true-RMS multimeter.

  Assessment – Apply a triangle wave from a signal generator to the input of a digital oscilloscope, then predict its RMS value based on peak measurements. Then, confirm the accuracy of that prediction using either the oscilloscope's RMS measurement capability or a true-RMS multimeter.

  Assessment – Apply a square wave from a signal generator to the input of a digital oscilloscope, then predict its RMS value based on peak measurements. Then, confirm the accuracy of that prediction using either the oscilloscope's RMS measurement capability or a true-RMS multimeter.

- **Outcome** – Apply the concept of phase shift to oscillographs

  Assessment – Given an oscillograph showing multiple AC voltage or current waveforms, measure the amount of phase shift between them; e.g. pose problems in the form of the "Phase shift from oscillographs" Quantitative Reasoning question.

- **Outcome** – Prove the concept of transformer turns ratios by experiment

  Assessment – Design and conduct an experiment to determine the turns ratio of a transformer from voltage measurements.

- **Outcome** – Independent research

  Assessment – Read and summarize in your own words reliable source documents on the history of electric power systems around the world. Recommended readings include *Networks of Power – electrification in Western Society, 1880-1930* by Thomas Parker Hughes.

  Assessment – Locate transformer datasheets and properly interpret some of the information contained in those documents including turns ratio, winding configuration, voltage ratings, current ratings, power ratings, etc.

Chapter 2

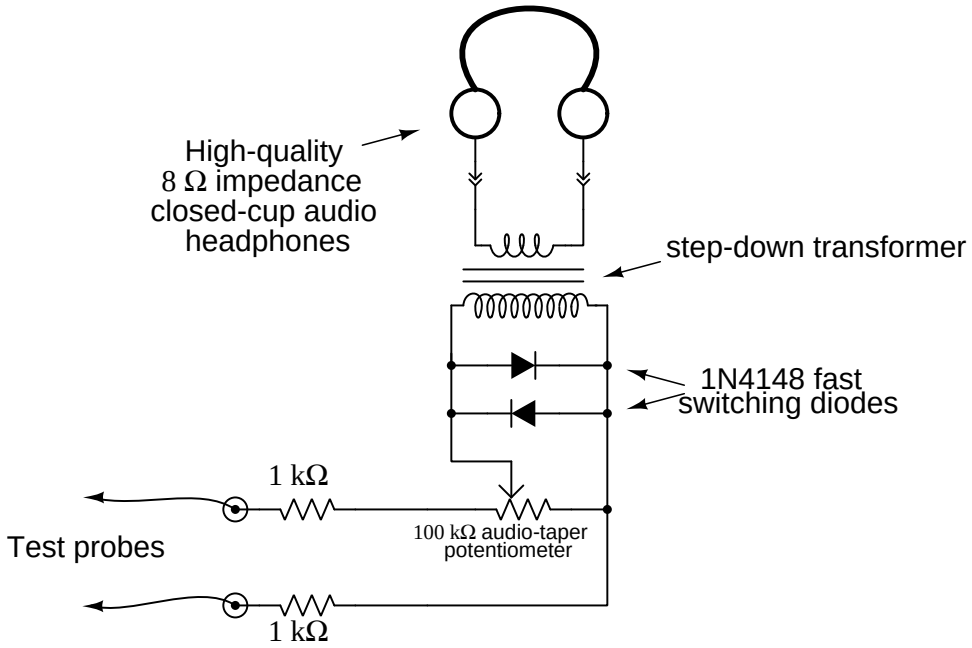
Case Tutorial

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: sensitive audio detector circuit

A useful project for sensing low-voltage signals is this "sensitive audio detector" circuit using headphones to convert electrical energy into acoustic energy you can hear:



Any AC signal with a frequency between approximately 20 Hz and 20 kHz (depending on the range of your own hearing) may be heard using this test instrument. Such an audio detector is extremely useful for probing audio amplifier circuits, allowing you to hear the signal at different points along the amplification circuitry. With good-quality headphones or earbuds, the sensitivity is great enough to allow the user to hear 60 Hz electrical "noise" even without direct contact of the probes (e.g. grounding one probe and holding the other *near* the noise source, or connecting both probes to an inductive pick-up coil). Even DC signals may be sensed by listening for a "click" or "scratching" noise whenever the test probes come into contact with a DC voltage source.

An interesting application of this test instrument is as a *null detector* in precision electrical measurement circuits such as Wheatstone bridges, where its purpose is to aurally indicate a *lack* of electrical potential between two points. To use this as a null detector in a DC circuit, you must connect one of the test probes in series with a pushbutton switch or some other convenient means of disconnection and reconnection, in order to make small DC voltages/currents audible. For example, when used to sense voltage between the measurement points of a Wheatstone bridge circuit it will indicate a state of balance when it produces no sound at all in response to being connected and disconnected from those points. Any time the bridge is not perfectly balanced, there will be an audible "click" in the headphones/earbuds when connected and disconnected from the circuit even with currents less than one microAmpere.

The transformer acts to match the low impedance of the headphones (typically 8 Ohms) to a much greater impedance presented at the high-turns winding, this impedance transformation described by the following formula:

$$\frac{Z_P}{Z_S} = \left(\frac{N_P}{N_S}\right)^2$$

Where,

Z_P = impedance at transformer's primary winding, in Ohms
Z_S = impedance at transformer's secondary winding, in Ohms
N_P = number of turns within transformer's primary winding
N_S = number of turns within transformer's secondary winding

An excellent choice for this transformer is a reclaimed power transformer from a common household microwave oven. When microwave ovens fail, it is usually the magnetron tube or some other component, not the power transformer. With microwave ovens being so common in American households, it should be easy to recover a power transformer from one if needed. Normally, this type of transformer steps up 120 Volts AC from the home's power wiring to approximately 2 kV AC necessary to energize the oven's magnetron tube. Here, we are using it as a step-down transformer, a microwave oven transformer's high-voltage ratings providing excellent isolation for safety in this application, ensuring no conductive connection whatsoever between the headphones and the circuit under test. It is recommended that you cut the connection between one end of the microwave oven's high-voltage winding[1] and the iron core of the transformer, so that the high-voltage winding is completely isolated from the frame.

A transformer with a large iron core such as a microwave oven power transformer also helps make very low-voltage DC signals audible by its ability to *store energy*. When connecting the detector circuit to a DC source, the transformer will build up a magnetic field of one polarity corresponding to the polarity of the applied voltage. When the connection to the source is broken (by lifting one or both of the test probes away from the point(s) of contact), this magnetic field rapidly collapses and in doing so delivers all that stored energy over a brief time interval to the only load still connected: the headphones. Energy, of course, is always conserved, but it is possible for the transformer in this context to release more *power* (i.e. energy transfer per unit time) than it receives. Thus, in addition to providing safe electrical isolation between the circuit under test and the headphones, as well as impedance transformation for weak AC signals, the transformer also augments weak DC signals by its ability to inductively store (and rapidly release) energy.

The two parallel-connected diodes serve to limit the maximum amount of voltage across the transformer's primary winding. These silicon model 1N4148 diodes have a typical forward voltage of approximately 0.7 Volts, meaning neither is able to conduct electricity unless the voltage across it reaches this value (in the correct polarity). Thus, any voltage less than 0.7 Volts (peak) will pass to the transformer's primary winding unattenuated, but any signal voltage greater than this will become "clipped" to a maximum value of 0.7 Volts. If the perceived volume at this peak signal amplitude is still too loud, one may insert a series resistance between the diode pair and the

[1] This is typically how microwave oven power transformers are wired: with the high-voltage winding connected at one end to chassis ground. Here, we neither need nor want this feature, but instead would prefer an isolated winding.

transformer's primary winding, experimenting with this resistor's value until a suitable maximum volume is found

A potentiometer provides simple volume control, useful for attenuating volume when sensing stronger signals. The two resistors in series with the test probes provide a minimum resistance between the test probes and transformer winding to prevent the transformer from excessively loading the circuit under test, as well as provide a maximum volume limit for listening. You may find it possible to use series resistors much greater than 1 kiloOhm each if your headphones are sensitive and your sense of hearing is acute. A good practice is to set the potentiometer to a position of minimal volume when first testing a signal of unknown magnitude, so that the resulting volume produced by the headphones will not be too loud. This is analogous to setting a manually-ranged multimeter to the highest range when testing a voltage or current, so that if the signal happens to be stronger than expected the meter's pointer will not be violently thrown to maximum position on its scale.

2.2 First-time oscilloscope experiment ideas

One of the most useful test instruments for electronic circuits is the *oscilloscope*, used to create a graphical plot of voltage over time. Oscilloscopes are extremely important for measuring AC (alternating) signals as well as transient (momentary) signals. When first learning the operation of oscilloscopes, it is informative to safely experiment with a real oscilloscope in order to become accustomed to its controls and functionality. Here is a list of experimental ideas that are safe, interesting, and rich with learning opportunities:

- **Measuring battery voltage** – connect an oscilloscope to the terminals of a battery, then interpret that battery's voltage using the divisions on the screen's vertical axis. Note that you will need to have the vertical coupling configured for "DC" in order for this to work!
- **Detecting ambient electric fields** – connect the oscilloscope's probe tip to a long wire and use that wire as a crude antenna to pick up AC electric fields with respect to Earth ground (the oscilloscope will already have an Earth-ground reference from its line power cord).
- **Detecting magnetic fields** – connect the oscilloscope's probe tip and ground clip to a wire coil, then place that coil nearby any AC line-powered device to plot the induced voltage. Note that the coil will be maximally sensitive if it has a large diameter and the multiple loops are close together (i.e. make a "flat" coil rather than a cylindrical coil).
- **Sensing electromagnetic induction** – connect the oscilloscope's probe tip and ground clip to a wire coil, then move a permanent magnet in and out of that coil's center while observing the voltage displayed by the oscilloscope. A large-diameter coil again will work best, and it is recommended to set the timebase to some slow value (e.g. 1 second or more per division) to most easily see the rising and falling voltage signal produced by the magnet's motion.
- **Measuring sound and vibration** – connect the oscilloscope to the terminals of a permanent-magnet loudspeaker, which is also able to function as a simple microphone. Speaking into the loudspeaker's cone will cause its diaphragm to vibrate and thereby produce a weak AC voltage measurable by the oscilloscope. Gently tapping your finger on the cone will produce voltage pulses easily displayed by the oscilloscope. If you configure the oscilloscope for "single-sweep' operation you will be able to capture hand-claps and other transient events.
- **Measuring a 1 kHz square wave signal** – most oscilloscopes provide a test terminal on their front panels outputting a 1 kHz square-wave signal useful for adjusting $\times 10$ probes. However, even with plain $\times 1$ probes this is still a useful test for learning to use an oscilloscope's basic functions. After locking the pulse waveform on the screen, use the vertical and horizontal divisions to verify the signal's magnitude, period, and frequency. The period (i.e. time per cycle) should be 1 millisecond if the frequency is 1000 Hz; i.e. a signal alternating at 1000 cycles per second will take just 1 millisecond to complete one cycle.

- **Measuring rotational speed** – connect the oscilloscope to the terminals of a small induction AC motor. Spinning the motor shaft will produce a small AC voltage due to the residual magnetism of the motor's rotor, and the frequency of this AC voltage will be directly proportional to the shaft's speed. A simple "shaded-pole" AC motor such as the type used for household fans works well for this purpose.

- **Measuring capacitor soakage** – connect the oscilloscope to the terminals of an electrolytic capacitor, a type well-known for a phenomenon called *soakage*. Configure the oscilloscope for "DC" coupling and a very slow timebase (i.e. multiple seconds per division). Short-circuit those terminals and watch the capacitor's voltage fall to zero. When you disconnect the shorting wire, however, you will notice the voltage value slowly rise over time due to this "soakage" effect, where some of the capacitor's stored energy was not dissipated during the short-circuit duration and slowly manifests itself at the capacitor's terminals.

- **Measuring light over time** – connect the oscilloscope to the terminals of a light-emitting diode (LED) which also functions as a simple photovoltaic cell when exposed to light. Point the LED's lens at a light source and measure the effect registered on the oscilloscope's display. This is particularly interesting if the light source happens to *pulse*!

- **Measuring temperature over time** – build a simple series voltage-divider network using one fixed-value resistor and a thermistor (i.e. temperature-dependent resistor) and energize this divider network with a suitable DC voltage source. Measure voltage across either the resistor or the thermistor using an oscilloscope configured for "DC" vertical coupling and set for a very slow timebase (e.g. 1 second per division). This will generate a plot representing temperature changes over time.

- **Measuring voltage noise in a simple DC motor circuit** – connect a DC motor to a battery or other suitable DC voltage source so that the motor runs continuously. Then, connect the oscilloscope probe terminals in parallel with that running motor. If configured for "DC" vertical coupling, you should see the elevated (DC) voltage powering the motor superimposed on a significant amount of jagged "noise" created by the motor's pulsing current draw. Setting the vertical coupling to "AC" allows you to zoom in on that noise while blocking the elevated DC voltage, so that the noise graph remains centered in the screen. You may also try connecting a capacitor in parallel with the running DC motor to see the effect it has in mitigating this voltage "noise".

- **Measuring current pulsations in a simple DC motor circuit** – insert a low-value (1 Ohm or less) *shunt resistor* in series with the DC motor described above, then connect the oscilloscope probe terminals in parallel with that shunt resistor. If configured for "DC" vertical coupling, you should see a graphical representation of that motor's pulsating current as it runs. Instantaneous current may be measured by taking the oscilloscope's voltage measurement at any point in time and dividing that value by the known shunt resistor value (i.e. using Ohm's Law $I = \frac{V}{R}$).

- **Measuring DC motor inrush circuit** – using the same battery-shunt-motor circuit described above, configure the oscilloscope for a slow timebase (e.g. 1 second or more per division) and then have is graph the motor's current when started and stopped by connecting/disconnecting the DC voltage source. Electric motors of all kinds typically draw high current when initially starting to rotate, their current tapering off over time as they come up to normal speed.

- **Measuring various signals in a 555 timer circuit** – consult the datasheet for a simple relaxation oscillator circuit based on a model 555 integrated circuit (IC), build the recommended "astable" test circuit on a solderless breadboard, and use the oscilloscope to measure various pulsating and oscillating voltage signals in the circuit. The same datasheet may also provide sample waveforms which you may compare with your measured oscillographs.

- **Measuring switch contact "bounce"** – connect the oscilloscope in parallel with a resistor that receives power from a DC voltage source through a switch. Configure the oscilloscope for "single-sweep" capture, and use it to display the transient "bouncing" of the switch's contacts as they open and close. Note that you will have to set the timebase to a fairly rapid sweep rate (e.g. 0.2 milliseconds per division or so) in order to successfully capture these rapid closure and opening events.

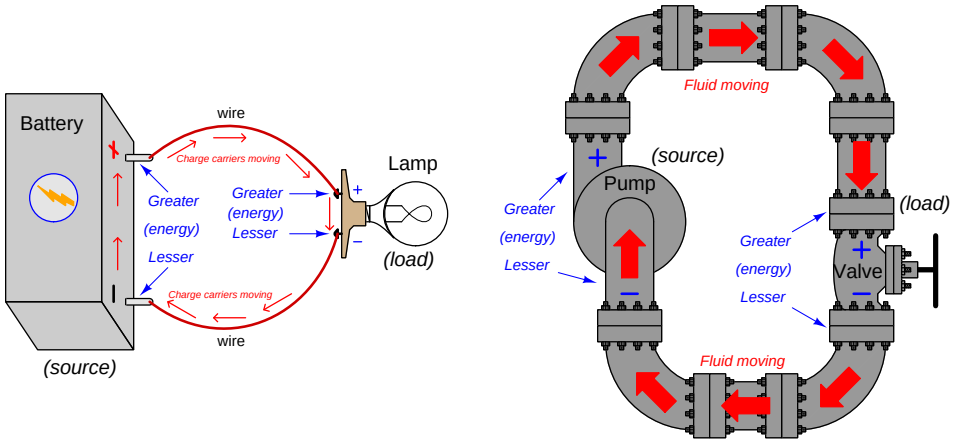
Chapter 3

Tutorial

3.1 AC versus DC

Energy is that which is able to set matter into motion, and it is known to exist in many different forms. One of those forms is electrical: imparting energy to mobile *electric charge carriers*, very similar to how molecules within a pressurized liquid possess energy. *Electricity* is the term we used to describe the transfer of energy to and from electric charge carriers, similar to the way we use the word *hydraulics* to describe the transfer of energy to and from liquids.

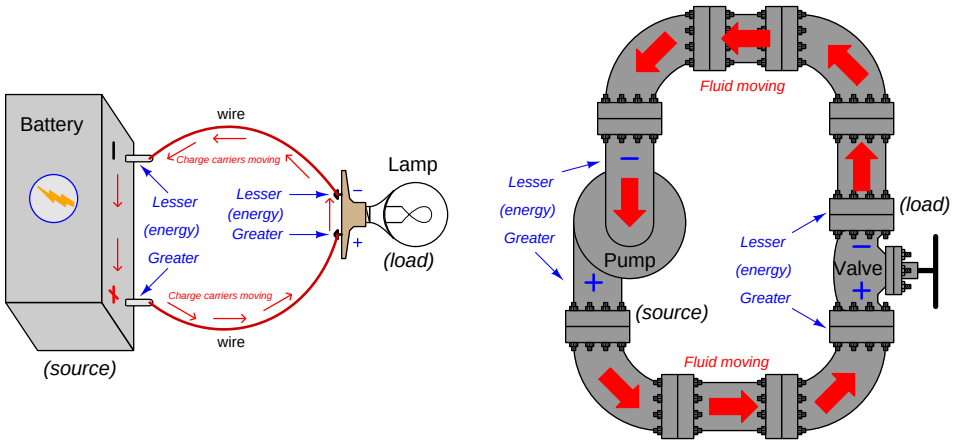
Voltage is the difference in energy experienced by an electric charge carrier if it moves (or were to move) from one physical location to another. The hydraulic equivalent of voltage is *pressure drop* or *pressure rise* from one location in a hydraulic system to another. When electric charge carriers flow from one location to another and experience a change in energy as a result, it means they have either acquired energy from an external *source* or delivered energy to an external *load*.



In the electric circuit (left), a battery imparts energy to electric charge carriers, taking them from a condition of lesser energy (−) to greater energy (+). Flowing through the wires and through the lamp, this *electric current* delivers energy to the lamp and in so doing the charge carriers suffer a loss of energy (from + to −). Energy is conserved, which in this case means the amount of energy lost by charge carriers through the lamp exactly equals the amount of energy delivered to them by the battery. Electric charge carriers *convey* energy from source to load in an electric circuit.

In the hydraulic circuit (right), a pump imparts energy to fluid molecules, taking them from a condition of lesser energy (−) to greater energy (+). Passing through the pipes and through the restriction valve, those fluid molecules deliver energy to the valve and in so doing suffer a loss of energy (from + to −). Energy is conserved, which in this case means the amount of energy lost by fluid molecules through the restriction valve exactly equals the amount of energy delivered to them by the pump. Fluid molecules *convey* energy from source to load in a hydraulic circuit.

If we reverse the source in each circuit, we find both the direction of flow and the +/− *polarity* symbols reverse as well, but each source remains a source and each load remains a load:



Each of these systems, in both orientations, delivers energy from source to load. Neither the electric lamp nor the hydraulic valve functions differently when direction of flow and voltage (pressure) polarities reverse. In fact, if the source in each circuit were replaced by one that periodically *oscillated* in direction, each system would continue to transfer energy from source to load, albeit in pulses rather than continuously.

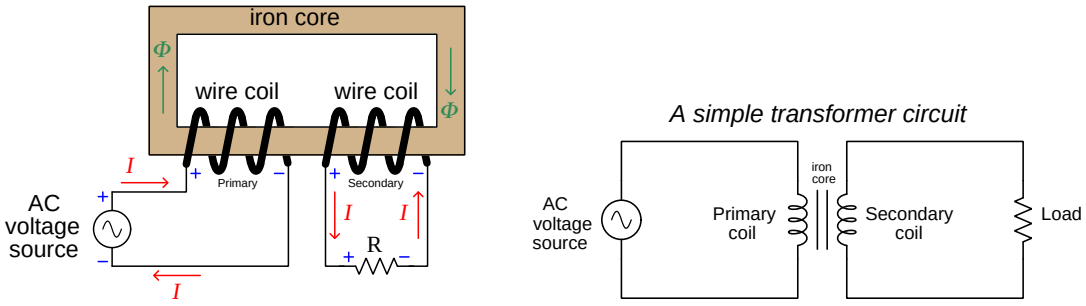
This is the essence of an *alternating current* circuit: one where the directions of flow and/or voltage (pressure) polarities regularly reverse.

Three major reasons explain the existence of "AC" electric circuits:

- Alternating-current (AC) generators and motors are simpler in construction and more reliable than direct-current (DC) generators and motors.
- Alternating-current (AC) works with devices called *transformers* that work to make electric power transmission more economical than direct-current (DC).
- If we use electricity as a *signal* to represent an oscillating parameter (e.g. sound waves, vibration, nerve impulses), then the signal must oscillate as well.

3.2 Transformers

Although the subject of *transformers* is complex enough to warrant its own tutorial, we will briefly examine them here as uniquely AC electrical components. In its simplest form a "transformer" consists of a pair of wire coils sharing a mutual magnetic loop, such that any magnetic field created by electric current passing through one coil must pass through the other coil as well. Such a device is shown in pictorial form (left) as well as schematic form (right):



If the electrical source energizing the first ("primary") coil outputs a varying current, the magnetic field produced by that coil must vary as well, and this variation over time *induces* voltage in the second ("secondary") coil as described by Faraday's Law of Electromagnetic Induction:

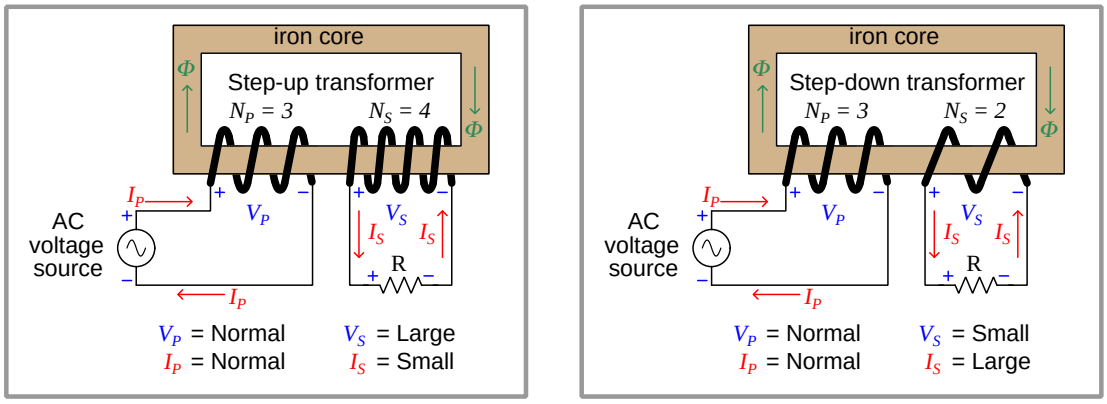
$$V = N\frac{d\Phi}{dt}$$

Where,

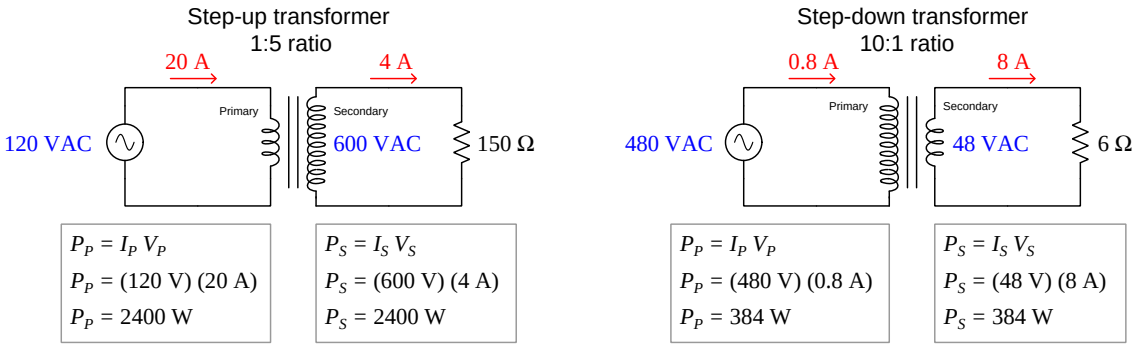
V = Voltage applied to the coil or induced by the coil (Volts)
N = Number of turns of wire
$\frac{d\Phi}{dt}$ = Rate of change of magnetic flux (Webers per second)

In order for this induction to occur, however, the magnetic field must vary over time, which is why transformers only function with AC and not with DC. Direct current – that is, electric current constant in amplitude and direction – would produce a constant magnetic field, and with no rate-of-change this magnetic field would not induce any voltage in the secondary coil. An *alternating* current, by its very nature, never ceases to change in strength and it is this ceaseless variation that makes a transformer work.

A useful consequence of this *mutual inductance* is the ability of a transformer to "step up" and "step down" AC voltage and current in accordance with the ratio of "turns" in each of the wire coils. A transformer with more turns of wire in the load (secondary) coil than in the source (primary) coil is called a *step-up transformer*. A transformer with fewer turns in its secondary than in its primary is called a *step-down transformer*:



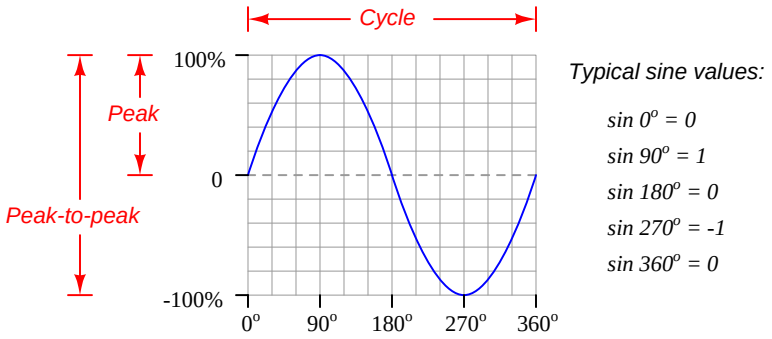
In accordance with the *Law of Energy Conservation*, the continuous power in the secondary circuit will be (nearly) equal to the continuous power in the primary circuit, any difference between the two power values being the result of dissipative losses in the transformer (e.g. wire resistance in the coils, magnetic hysteresis in the core, both converting some of the electrical energy into heat). An ideal transformer outputs exactly as much power as it takes in from the source, as illustrated by these examples:



As you might imagine, an electrical component having no moving parts that is able to convert high-voltage to low-voltage (and vice-versa) is very useful, especially for electrical power systems. The fact that these devices require AC and cannot function on DC is one of the most significant historical reasons why AC triumphed over DC for electrical power distribution.

3.3 AC quantities

Voltage and current in a steady-state direct-current (DC) circuit are easy to quantify. Voltage is simply the energy difference between two locations in the circuit per quantity of charge carriers[1], while current is the rate of charge carrier flow[2]. In an alternating-current (AC) circuit, however, these values constantly shift in amplitude and in direction. Rather than specify a constant value for either, we must express these as *waves* oscillating over time. Shown below is a *sine wave* as plotted on a test instrument known as an *oscilloscope*[3], so named because its shape is defined by the trigonometric *sine* function, typical of the voltage induced in the coils of an AC generator (also called an *alternator*) as its shaft rotates through the angles of a full circle:



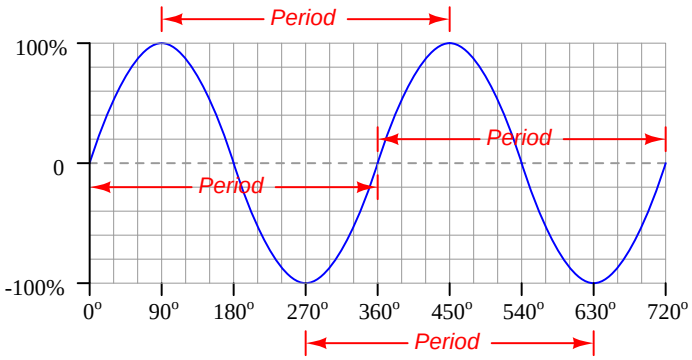
By definition, one *cycle* of a wave is 360 degrees of revolution, during which the wave will reach both a positive peak value as well as a negative peak value. *Peak* value, in fact, is a simple method of quantifying AC amplitude. Sometimes it is useful to express an AC quantity in terms of its over-all wave height from positive peak to negative peak, and we call this the *Peak-to-Peak* amplitude of the wave.

A pleasant discovery in resistive AC circuits is that values of voltage and current at any given instant in time relate to one another by all the same laws and principles learned for DC resistor circuits. For example, if a 3 Ohm resistor has an alternating current passing through it, the moment in time when that current reaches a value of 4 Amperes the voltage drop across that resistor will be 12 Volts in accordance with Ohm's Law. Joule's and Kirchhoff's Laws (both voltage and current) similarly apply at any specific instant in time within an AC circuit, for example:

- 150 mA (pk) through a resistor dropping 3 V (pk) gives a peak power of 450 mW
- Two series-connected AC voltage sources, one peaking at 5 V and the other peaking at 2 V simultaneously, create a total voltage of 7 V (pk).
- Two parallel-connected AC current sources, one peaking at 200 mA and the other peaking at 300 mA simultaneously, create a total current of 500 mA (pk).

[1] One *Volt* of voltage is equivalent to one *Joule* of energy difference per *Coulomb* of charge carriers.
[2] One *Ampere* of current is equivalent to one *Coulomb* of electric charges moving past a point every *second* of time.
[3] An oscilloscope is simply a graphical voltmeter, plotting voltage (vertical axis) as a function of time (horizontal axis). The plot generated by an oscilloscope is called an *oscillograph*.

A feature uniquely distinguishing AC from DC is the *periodic* nature of the former: AC voltages and currents repeatedly *vary* in amplitude and *alternate* in direction. The amount of time required for the voltage or current to come "full-circle" to repeat the same pattern is called the *period* of that wave. By definition, period is the amount of time equivalent to 360 degrees of rotation in the wave's evolution, and may be measured horizontally between any two points on the wave representing a repetition of the cyclic pattern:



Period is typically measured in seconds, and the reciprocal of period is called *frequency* (f) which used to be measured in *cycles per second* (cps) but now has its own unit of the *Hertz* (Hz)[4]. For example, in many parts of the world the standard frequency for AC power systems is 60 Hz, equating to a period of 16.67 milliseconds. In other parts of the world the standard frequency for AC electric power is 50 Hz, with a period of 20 milliseconds.

If an alternating quantity happens to move through space with a known velocity, as is the case with electromagnetic waves (moving at the speed of light) and with acoustic vibrations (moving at the speed of sound[5]), we may describe the period of the wave in terms of physical *distance* as well as time. The distance covered by a propagating wave over one whole cycle (i.e. traveling for a time duration of one period) is called the *wavelength* and is symbolized by the Greek letter "lambda" (λ):

$$\lambda f = c$$

Where,

λ = Wavelength in meters

f = Frequency in cycles per second, or Hertz

c = Propagation velocity, approximately 3×10^8 meters per second for electromagnetic waves

For example, radio waves emanating from a transmitting antenna operating at a frequency of 99.1 MHz will have a wavelength of 3.027 meters.

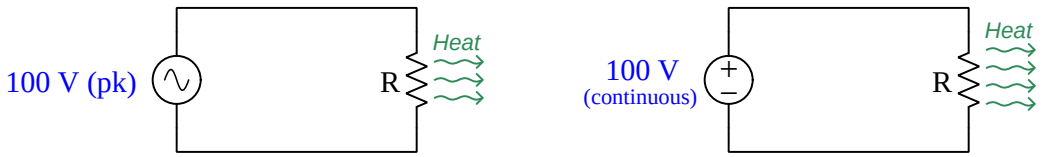
[4] Alternatively, frequency may be expressed as *inverse seconds*, e.g. 60 s^{-1} instead of 60 cps or 60 Hz. Yet another way of expressing the frequency of an AC quantity is in terms of the number of *radians per second* covered by the evolving wave. This is called *natural frequency* or *angular velocity*, and it is symbolized as a lower-case Greek letter "omega" (ω) instead of f.

[5] The speed of sound varies with the type of material the sound waves must propagate through, and as such it is not a fixed limit. Similarly, the speed of light varies with material as well, but at least with the speed of light we have a maximum velocity defined through a perfect vacuum: approximately 3×10^8 meters per second.

3.4 Quantifying AC voltage and current

Alternating current (AC) was first applied to early electrical *power* systems in order to realize certain efficiencies that were impossible for DC with the technology of the day. As such, it became necessary to rate AC voltage and current in terms that could be equitably compared to DC. In an electric power system, the purpose is to use electricity as an *energy-delivery medium*, and so a natural basis for comparison between AC and DC was to rate AC in terms of equivalent energy-delivery with DC.

This concept may not be obvious at first, and so some elaboration is helpful. Imagine setting up an AC sine-wave voltage source with a peak amplitude of 100 Volts so that it energizes a resistor, and also setting up a DC voltage source of 100 Volts connected to an identical resistor. Which of these two resistors will dissipate more energy in the form of heat over an identical span of time?

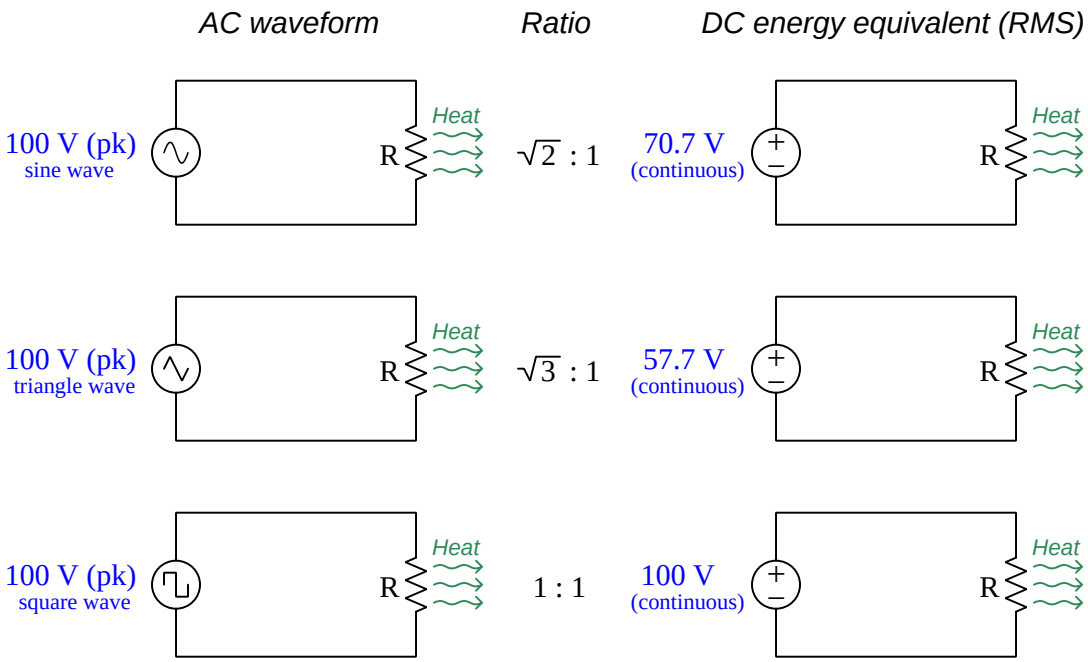


It should be clear to see that the DC-powered resistor will dissipate more heat, because the DC source outputs 100 Volts *continuously* while the AC source only outputs 100 Volts *intermittently*. Simply put, the DC source *works harder* than the AC source given equal peak voltage value. Therefore, *peak* amplitude is a misleading indicator of AC voltage or current when compared to DC on the basis of work done.

If the AC source happens to output a sinusoidal (i.e. sine-wave-shaped) voltage, the ratio between the power dissipated by the resistor at 100 Volts DC versus the power dissipated by an identical resistor at 100 Volts (peak) AC is $\frac{\sqrt{2}}{2}$, which is approximately 0.707. Thus, our 100 Volt (pk) AC voltage is equivalent in work value to a 70.7 Volt DC source. Conversely, the AC source would have to reach peak voltage of 141.4 Volts (peak) to achieve the same energy-delivery effectiveness as a 100 Volt DC source. We refer to this DC-equivalent value as the *Root-Mean-Square*[6] or *RMS* value of the AC waveform.

[6]This strange-sounding designation comes from the mathematical steps necessary to calculate the equivalence. First, we need to compute the instantaneous power dissipated by the load at multiple points in time within one cycle of the source's AC voltage or current. Whether we are working with voltage or current, computing power for a given resistance involves a quadratic (*square*) function: $P = I^2R$ or $P = \frac{V^2}{R}$. After calculating a set of instantaneous power values throughout one cycle, we average (i.e. *mean*) those values together to simulate the same total amount of energy delivered by a steady (DC) voltage or current. Last, we take the average power delivered to the load and compute from that the equivalent DC voltage or current necessary for that amount of power given the original load resistance value, which necessarily involves a square-*root* function: either $V = \sqrt{PR}$ for voltage or $I = \sqrt{\frac{P}{R}}$ for current. Therefore, the DC-power-equivalent value of an AC voltage or current is the *root* of the *mean* of the *square* for a set of instantaneous values within the AC cycle, or *RMS*. The *Programming References* chapter of this learning module shows how these calculations may be performed using the C++ programming language, beginning on page 69 of section 5.6.

The conversion ratio between peak and RMS values for any AC waveform depends strongly on the particular *shape* of that waveform. For sinusoidal waves the conversion factor is $\sqrt{2}$, but it is not the same for other wave-shapes:



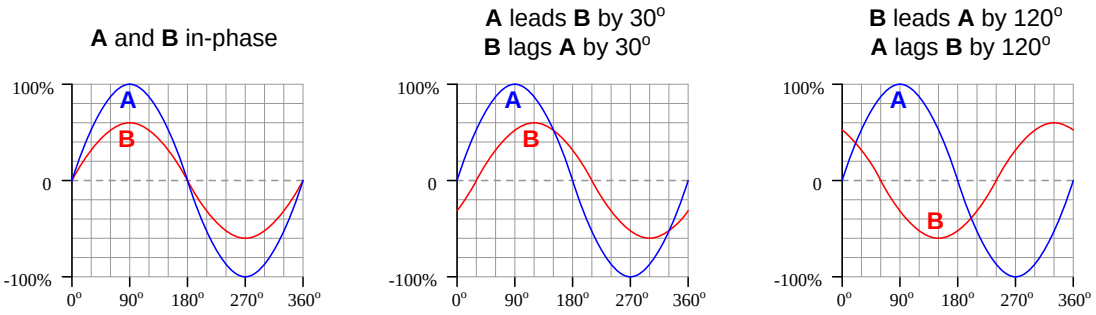
Given equal peak amplitudes, the "triangle" wave-shape is weaker than the sine wave, while a square wave is stronger than a sine wave. The differences are easy to understand if you carefully examine the wave-shapes and see which shape spends more time closer to its peak value throughout the cycle than another.

For the reasons stated earlier, voltage and current values in AC electric power circuits are always specified in RMS units. This poses a problem for AC voltmeters and ammeters tasked with reporting voltage and current values in RMS terms, because neither analog meter mechanisms nor digital electronic circuits naturally respond to the *heating* power of an AC signal. Clever circuits exist to compute RMS values, and the meters employing them are called *true-RMS* instruments, but these circuits are relatively expensive to manufacture compared to circuits responding either to the average or peak values of an AC signal. Analog meters and cheap digital meters typically respond to the averaged (mean) absolute-values of an AC waveform, which is not equal to the RMS value[7], and so these non-RMS meters are intentionally mis-calibrated so as to register RMS values *when measuring sinusoidal voltages and currents*. If measuring some other wave-shape, these non-RMS meters will not accurately register the actual RMS value of the AC signal.

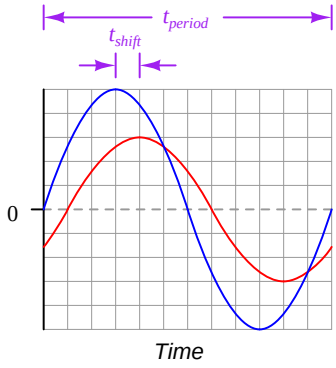
[7] The RMS value of an AC waveform is based on the square-root of the averaged instantaneous power values, each of those derived from the square of the instantaneous voltage or current values within an AC cycle. The "average" of a waveform's absolute-valued instantaneous amplitudes is not the same, as it omits the squares and the square-roots.

3.5 Phase shift

When two AC waveforms having the same frequency (i.e. same period) are compared against one another in the time-domain, we may express the difference in angle between them as a quantity called *phase*. Two waves that are perfectly synchronized with each other are said to be *in phase*, while two waves out of step with one another are said to be *out of phase* or *phase-shifted*. Some examples of sine wave pairs appear below:



When two waveform reach their respective peaks at different times, the waveform that's ahead is said to be *leading* and the waveform that's behind is said to be *lagging*. Quantifying phase shift is as easy as measuring the difference in time between respective features (e.g. between positive peaks, between negative peaks, or between zero-crossing points of similar slope) and comparing that to the period, setting that quotient equal to a ratio between phase shift (θ) and 360^o:



$$\frac{t_{shift}}{t_{period}} = \frac{\theta}{360^o}$$

Like voltage, phase is a relative quantity: it exists only as a *difference* between two or more waveforms. Often the voltage waveform of the circuit's source is taken to be the "reference" wave against which the phase angle of all other voltages and currents in the circuit are measured. Phase shift is a fixed quantity when two or more waves exist at the same frequency, but phase continuously varies between waves of differing frequency.

Chapter 4

Derivations and Technical References

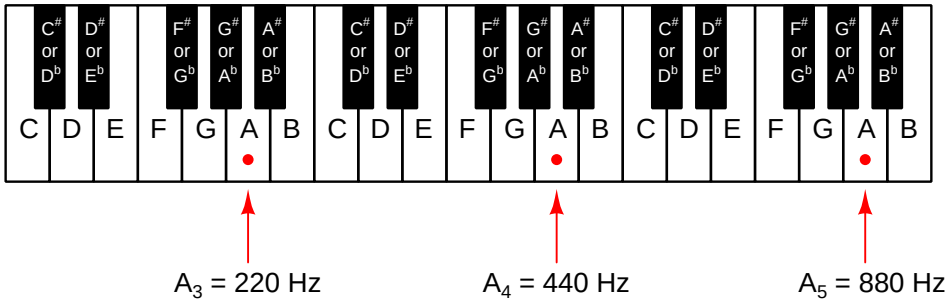
This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

4.1 Musical pitch

In musical terms, frequency is equivalent to *pitch*. Low-pitch notes such as those produced by a tuba or bassoon consist of air molecule vibrations that are relatively slow (low frequency). High-pitch notes such as those produced by a flute or whistle consist of the same type of vibrations in the air, only vibrating at a much faster rate (higher frequency).

An international standard exists to define the pitch of tones used in Western music, using seven letters (A through G) and modifying characters called *sharps* and *flats*. Furthermore, the *scientific pitch notation* scheme uses numerical subscripts to differentiate between repeated letters. Every repetition of letters in the sequence represents a doubling of pitch called an *octave*: A_4 is standardized[1] at 440 Hz, which means the next "A" in increased pitch (symbolized as A_5) must be 880 Hz. The "A" pitch below A_4 (symbolized as A_3) is 220 Hz. There is rationale for the repeated lettering of pitches: two pitches bearing the same letter (e.g. A_3 and A_4) sound very much alike to the human ear.

Counting all the white and black keys on a piano keyboard, we see there exist twelve pitches starting at any particular tone and ascending or descending until just before reaching a similarly-labeled tone. Thus, each octave (i.e. doubling of frequency) is divided into twelve distinct pitches in the Western musical system:



In the *equal-temperament* pitch system, each interval in the twelve-step division represents the same ratio of frequencies. Specifically, we may mathematically predict the next pitch in a twelve-tone equal-temperament system using the following formula:

$$f_n = f_0 2^{\frac{n}{12}}$$

Where,

f_0 = Frequency (pitch) of the starting tone

f_n = Frequency (pitch) of the tone n steps above

If one octave spans twelve contiguous keys on a piano keyboard, then the frequency of the next-octave higher pitch must be $2^{\frac{12}{12}}$ which is 2^1 or double the starting frequency.

[1] Perhaps the word *standardized* is too strong a term to describe the cacophony of pitch "standards" found in the musical world. Some orchestras tune their instruments such that A_4 is something other than 440 Hz (e.g. 441 Hz, 442 Hz), and some types of instruments (notably, keyboard percussion instruments such as vibraphone, glockenspiel, and marimba) are usually manufactured with A_4 set to 442 Hz. Melody and harmony are both defined by the *ratios* of pitch frequencies, and so musical pieces are still readily identified and enjoyed when played on other "standard" tunings. Problems arise, however, when instruments tuned to disparate pitch standards play together.

The following table lists equal-tempered tones and pitch frequencies between A_4 and A_6:

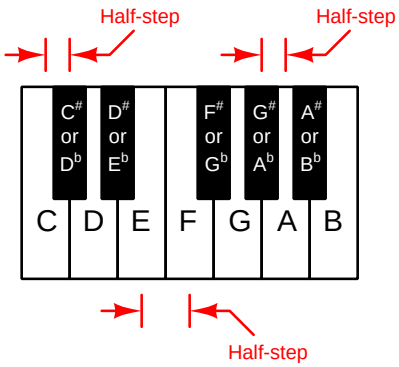
Letter symbol	**Mathematical expression**	**Pitch (Hz)**
A_4	$440 \times 2^{(0/12)}$	440.00
$A_4^\sharp$ or $B_4^\flat$	$440 \times 2^{(1/12)}$	466.16
B_4	$440 \times 2^{(2/12)}$	493.88
C_5	$440 \times 2^{(3/12)}$	523.25
$C_5^\sharp$ or $D_5^\flat$	$440 \times 2^{(4/12)}$	554.37
D_5	$440 \times 2^{(5/12)}$	587.33
$D_5^\sharp$ or $E_5^\flat$	$440 \times 2^{(6/12)}$	622.25
E_5	$440 \times 2^{(7/12)}$	659.26
F_5	$440 \times 2^{(8/12)}$	698.46
$F_5^\sharp$ or $G_5^\flat$	$440 \times 2^{(9/12)}$	739.99
G_5	$440 \times 2^{(10/12)}$	783.99
$G_5^\sharp$ or $A_5^\flat$	$440 \times 2^{(11/12)}$	830.61
A_5	$440 \times 2^{(12/12)}$	880.00
$A_5^\sharp$ or $B_5^\flat$	$440 \times 2^{(13/12)}$	932.33
B_5	$440 \times 2^{(14/12)}$	987.77
C_6	$440 \times 2^{(15/12)}$	1046.50
$C_6^\sharp$ or $D_6^\flat$	$440 \times 2^{(16/12)}$	1108.73
D_6	$440 \times 2^{(17/12)}$	1174.66
$D_6^\sharp$ or $E_6^\flat$	$440 \times 2^{(18/12)}$	1244.51
E_6	$440 \times 2^{(19/12)}$	1318.51
F_6	$440 \times 2^{(20/12)}$	1396.91
$F_6^\sharp$ or $G_6^\flat$	$440 \times 2^{(21/12)}$	1479.98
G_6	$440 \times 2^{(22/12)}$	1567.98
$G_6^\sharp$ or $A_6^\flat$	$440 \times 2^{(23/12)}$	1661.22
A_6	$440 \times 2^{(24/12)}$	1760.00

It is important to understand that while 440 vibrations per second (i.e. 440 Hz) is often considered the "standard" pitch for A_4, this is a convention and not a physical necessity. In fact the "standard"

pitch for "A" has indeed changed significantly in the history of Western music. This fact points us toward an essential truth of music: *relative pitch is more important than absolute pitch.* If an orchestra de-tuned all their instruments such that every musician's "A" was 430 Hz instead of 440 Hz, and all the other notes were proportionately shifted (i.e. every pitch now being $\frac{430}{440}$ of its normal value), anyone could still recognize and appreciate the songs played. Melodies would still sound correct, and harmonies would still be harmonious. Perhaps there would be a few unique individuals who could tell the difference and would be irritated by it, but to the rest of us it would still be beautiful music.

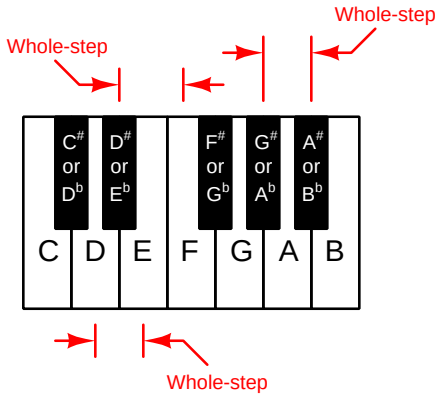
In honor of this fact, much of music theory is oriented around the concept of an *interval*: the relative "distance" between different pitches. Mathematically, an interval is a *ratio* of two pitches to each other. For the interval called an "octave," the ratio is 2:1 (or powers of 2:1, such as 4:1 or 8:1).

The pitch-distance between any two adjacent keys on a piano keyboard is called a *half-step* or a *semitone*. Mathematically, it is equal to a pitch ratio of $2^{1/12}$, or approximately 1.0595. To illustrate, the interval separating C from $C^\sharp$ is a half-step. So is the interval separating $G^\sharp$ from A. So is the interval separating E from F:



There are 12 such half-steps, or semitones, in the span of one octave. You can see this for yourself by starting at any key on a piano keyboard and counting every key either upward or downward until you reach the next octave of the original key: 12 steps are necessary to reach the octave pitch. You can also see this by referencing the previous table of pitches, counting the number of steps to go from A_4 to A_5, or any other octave span.

A *whole step*, also called a *whole-tone*, is simply the span covered by two half-steps, mathematically equal to a pitch ratio of $2^{2/12}$, or approximately 1.1225. The pitch-difference between D and E is a whole step. So is the the interval separating $G^{\sharp}$ from $A^{\sharp}$. So is the interval separating $D^{\sharp}$ to F:



Music theorists give names to all the different intervals up to and including the octave:

Interval name	**Symbol**	**Pitch ratio**	**Whole+Half steps**
Perfect Unison	P1	$2^{0/12}$	(no steps)
Minor Second	m2	$2^{1/12}$	1 half-step
Major Second	M2	$2^{2/12}$	1 whole-step
Minor Third	m3	$2^{3/12}$	1 whole-step + 1 half-step
Major Third	M3	$2^{4/12}$	2 whole-steps
Perfect Fourth	P4	$2^{5/12}$	2 whole-steps + 1 half-step
Augmented Fourth Diminished Fifth	aug4 dim5	$2^{6/12}$	3 whole-steps a.k.a. a *"tri-tone"* interval
Perfect Fifth	P5	$2^{7/12}$	3 whole-steps + 1 half-step
Minor Sixth	m6	$2^{8/12}$	4 whole-steps
Major Sixth	M6	$2^{9/12}$	4 whole-steps + 1 half-step
Minor Seventh	m7	$2^{10/12}$	5 whole-steps
Major Seventh	M7	$2^{11/12}$	5 whole-steps + 1 half-step
Perfect Octave	P8	$2^{12/12}$	6 whole-steps

Some of these intervals are very nearly *rational* in the mathematical sense of the word: a ratio of two integers. For example, the major third interval ($2^{4/12}$) is very nearly equal to the fraction

$\frac{5}{4}$; the perfect fourth ($2^{5/12}$) very nearly equal to the fraction $\frac{4}{3}$; and the perfect fifth ($2^{7/12}$) very nearly equal to $\frac{3}{2}$. Some tuning systems deviating from equal-temperament (e.g. *just* temperament being one of them) actually fix these ratios at their rational values, the purpose being to achieve slightly more precise harmony when those pitches are played together.

Chapter 5

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*.

5.1 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.

5.2 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.

5.3 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`.

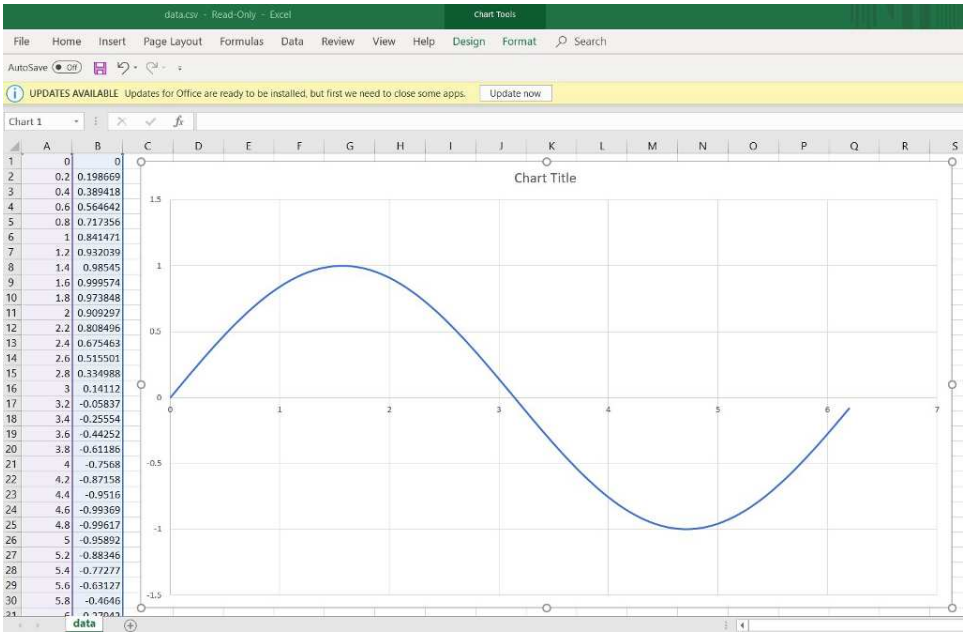
As previously mentioned, C++ lacks a standard library of graphics functions for plotting curves and other mathematical shapes to the computer's screen, which is why we opted to use *standard console* characters to do the same. If a truly *graphic* output is desired for our waveform plot, there are relatively 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.

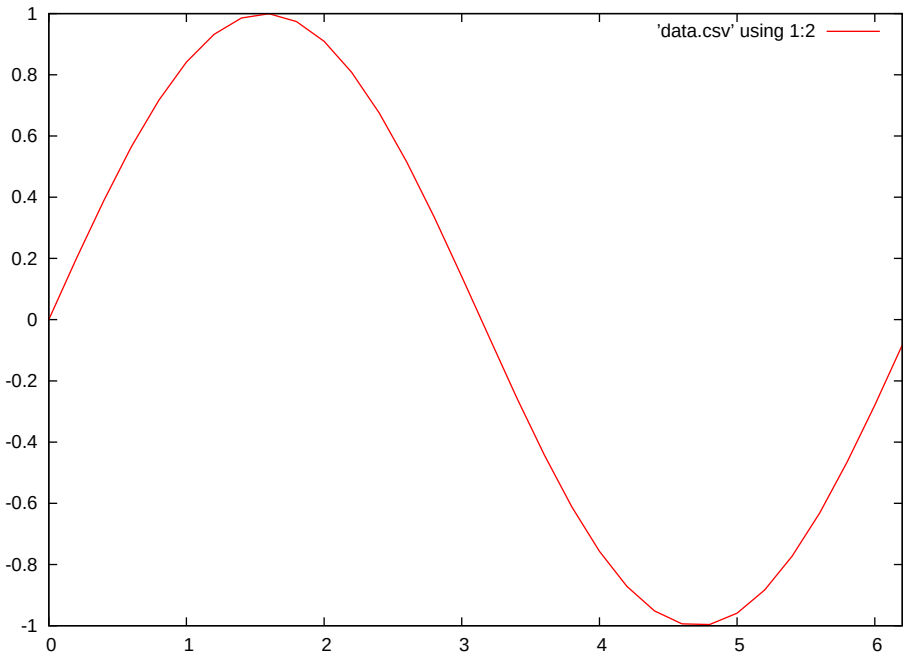


[11] A relatively easy way to do this is to run the C++ program from a console, using the *redirection* symbol (`>`). For example, if we saved our source code file under the name `sinewave.cpp` and then entered `g++ -o sinewave.exe sinewave.cpp` at the command-line interface to compile it, the resulting 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`).

5.4 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.

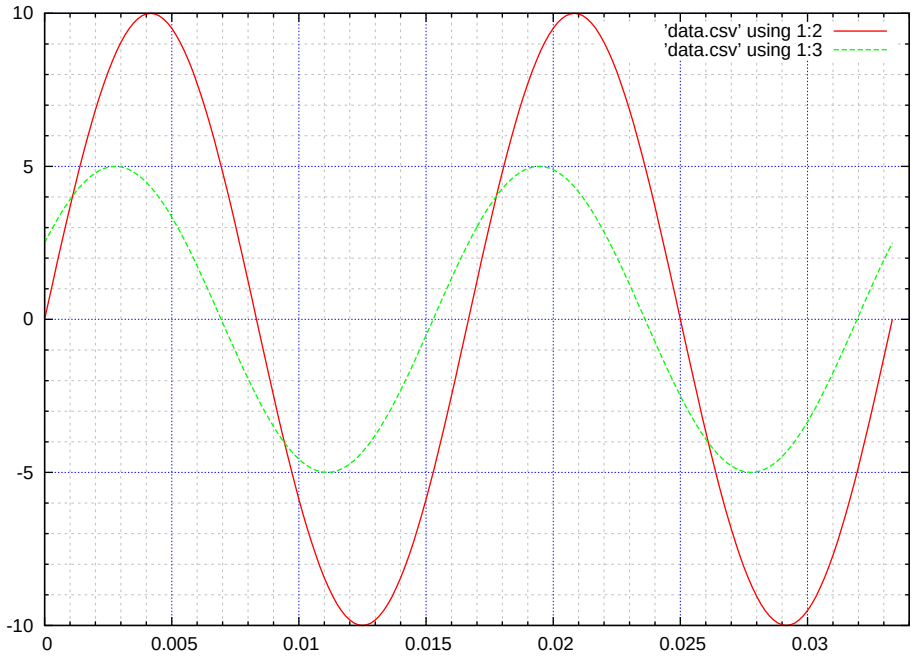
The sine functions are computed within these last `cout` instructions. The product of time and frequency (*seconds* times *cycles per second*) yields a result in *cycles*. Phase shift was entered in *degrees*, so division by 360 is necessary to case phase shift into cycles because there are 360 degrees 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:

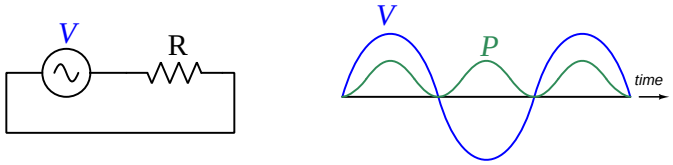
```
phaseplot.exe | tee data.csv
```

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.

5.5 Using C++ to compute RMS value

The *root-mean-square* or *RMS* value of an AC voltage or current waveform is the DC equivalent voltage or current that would dissipate the same power in an identical resistive load. Most AC voltages and currents are rated by their RMS values rather than their peak values, especially in electric power circuit, for the sake of equivalence between AC and DC. For example, in the United States the standard residential receptacle voltage is 120 Volts RMS, which means a resistive heating element connected to that source would dissipate precisely the same amount of power – averaged over time – as it would connected to a 120 Volt DC source.

Since alternating current (AC) varies in magnitude over time, the amount of power dissipated by a resistive load connected to an AC source must also vary from one instant to another. By contrast, a resistive load connected to a regulated DC source dissipates the same amount of power from one moment to another. Assuming a sinusoidal AC voltage energizing a resistor, and calculating instantaneous power using the equation $\frac{V^2}{R}$, the voltage and power waveforms would appear as follows:



Note how power is always a positive quantity even as voltage oscillates between positive and negative values, owing to the *squared* term in the power equation.

In order to compute the equivalent DC voltage value (i.e. resulting in the same average power dissipation at the resistor) we must calculate the total amount of energy dissipated by that resistor over some period of time encompassing at least one whole cycle of the AC voltage, then calculate the average DC power as energy divided by time[14]. Once we know the average power for the circuit, we may use the same power equation to solve for the equivalent DC voltage ($V = \sqrt{PR}$).

Note the square and square-root operations used in this definition of equivalent DC voltage: when computing instantaneous resistor power at any point in the AC voltage's waveform we must square that voltage quantity and then divide by resistance. When computing equivalent DC voltage at the end we must square-root the power and resistance values. Between those two steps we take the mathematical average (i.e. the *mean*) of the AC power values based on total energy dissipated over a specified time period. Thus, the power-equivalent DC voltage is the square root of the mean of the square of the instantaneous AC voltage values: *root-mean-square*, or *RMS*.

[14] *Energy*, of course, is that which sets matter into motion. *Power* is the rate at which energy transfers over time ($\frac{dE}{dt}$).

Shown here is the listing of a C++ program performing all these calculations. Values of AC peak voltage, resistance, time period, and time step interval are embedded in the code, so that when it is compiled and run it simply reports total energy, equivalent DC power, and equivalent DC voltage for the simulation:

```
#include <iostream>
#include <cmath>

using namespace std;
float wave (float);

int main (void)
{
  float t, v, p, dcv, dcp;
  float t_int = 0.0001;
  float t_end = 1.0;
  float freq = 60.0;
  float vpk = 100.0;
  float e = 0.0;
  float r = 1.0;

  for (t = 0; t <= t_end; t = t + t_int)
    {
      v = vpk * wave (360 * freq * t);
      p = pow (v, 2) / r;
      e = e + (p * t_int);
    }

  dcp = e / t_end;
  dcv = sqrt (dcp * r);

  cout << "Total energy dissipated = " << e << " Joules" << endl;
  cout << "DC equivalent power = " << dcp << " Watts" << endl;
  cout << "DC equivalent voltage = " << dcv << " Volts (RMS)" << endl;

  return 0;
}

float wave (float degrees)
{
  float radians;
  radians = degrees * M_PI / 180.0;
  return sin (radians);
}
```

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 (`=`)
- Loops (`for`)
- Basic arithmetic (`*`, `/`)
- Arithmetic functions (`sin`, `pow`, `sqrt`, modulus (`%`))
- Printing text output (`cout`)
- Custom functions: prototyping, return values, arguments
- Conditionals (`if`)
- Casting variable types

Studying the source code in the order it will be executed by the computer (left to right, top to bottom), we see the `#include` and `namespace` lines telling the C++ compiler software how to interpret many of the instructions that follow. We also see the `main` function enclosing all of our simulation code. The line `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. C++ ignores whitespace, and so all indentation of lines is done merely to make the source code easier for human eyes to read.

Within the `main` function we see a collection of *declarations* for floating-point variables used in this program. Each line terminates with a semicolon character (`:`) which is necessary for a language such as C++ that ignores whitespace and therefore must be told where each instruction line ends. Some of these lines also *initialize* the variable (i.e. assign a numerical value to it) at the time of declaration. These initializations are where we see the "hard-coded" values for time interval (`t_int`), ending time (`t_end`), frequency (`freq`), peak AC voltage (`vpk`), total dissipated energy (`e`, starting at a value of zero), and resistance (`r`). For this particular simulation, we are set to calculate at time intervals of 0.1 milliseconds each, for a period of 1 second, at a frequency of 60 Hz, a peak AC voltage value of 100 Volts, while energizing a 1 Ohm resistor.

Next comes a `for` loop, instructing the computer to repeatedly execute code while some condition is met. In this case, the condition is the value of time (`t`), starting at zero and increasing by

increments of `t_int` until the end time (`t_end`) is reached. Within the curly-braced boundaries of this `for` loop are three instructions. The first computes the value of a sine wave function with a peak value of `vpk` (more details on this to follow, soon!), the second calculates instantaneous power dissipated by the resistor at that voltage value using Joule's Law ($P = \frac{v^2}{R}$), and the third tallies a running total of energy dissipated by the resistor (i.e. instantaneous power `p` multiplied by the time interval `t_int`).

After the `for` loop finishes looping through those three instructions, we are left with a value for total energy (`e`) dissipated by the resistor over the time period spanning from `t = 0` to `t = t_end`. The next instruction computes the average amount of power for this circuit, equal to the power of a DC source doing the same amount of average work over time as this AC source, taking total energy and dividing it by total simulated time (`dcp = e / t_end`). After this equivalent DC power is computed, the next line calculates the equivalent amount of DC voltage needed to dissipate this much power (`dcv = sqrt(dcp * r)`) using Joule's Law ($V = \sqrt{PR}$)

Finally, the program concludes with three `cout` instructions printing data to the console where we may view it. First, we show total energy dissipated by the resistor over the simulated time period, then the equivalent DC power for that same period, and finally the equivalent (RMS) DC voltage. The last instruction within the `main` function returns an integer value of zero, not necessary for program execution but merely a programming convention of C and C++.

As promised previously we will now explore how the sine function gets computed in this program. Below the *main* function we see a custom function named `wave` used to compute the sine of an angle. This function was prototyped in a line near the beginning of the code listing, but the actual function with its own lines of code are at the very bottom of the listing. When this function is "called" by an instruction within `main`, the computer's flow of execution jumps down to the `wave` function to execute its instructions, and then returns back to `main` with a floating-point value of the sine function in hand. The sole purpose of the `wave` function is to take in an "argument" for the value of an angle in degrees, convert that into radians which the C++ `sin` function requires, compute the sine of that angle, and return the result.

This program would have worked quite nicely with all the angle-conversion and sine instructions written within the `main` function, but there is a very practical reason for containing these instructions within their own custom function called `wave`: it allows us to easily re-write the `wave` function to describe some waveform that is not a sine wave, without altering any lines of code within the `main` function. This sort of *modular programming* is a very good habit to cultivate, as it makes large and complex programs easier to manage, as well as facilitates re-use of code in future projects.

It's time to actually *run* this program and evaluate its results:

```
Total energy dissipated = 4999.73 Joules
DC equivalent power = 4999.73 Watts
DC equivalent voltage = 70.7088 Volts (RMS)
```

According to the simulation, an AC sinusoidal source of 100 Volts and 60 Hertz powering a 1 Ohm load resistance for 1 second will expend approximately 5000 Joules of energy, equivalent to a 70.7088 Volt DC source powering the same size load resistance for the same amount of time. The RMS figure of 70.7088 Volts agrees with the theoretical peak-to-RMS conversion factor of $\frac{\sqrt{2}}{2}$ used for sinusoidal wave-shapes.

If we edit the source code for our program to have a total time (`t_end`) of 5 seconds instead of 1 second, we see the total energy figure increase, but no substantial change in either DC equivalent power or RMS voltage:

```
Total energy dissipated = 25002.4 Joules
DC equivalent power = 5000.48 Watts
DC equivalent voltage = 70.7141 Volts (RMS)
```

Ideally the DC equivalent power and voltage would not be affected at all by an extension of time, but slight rounding errors in the computer's arithmetic add up to make these tiny variations.

Now that we have seen how this program successfully computes the RMS value of a peak-specified sinusoidal voltage, let's explore other wave-shapes. A *square wave* is relatively easy to simulate, by returning a value of +1 from if the angle is less than 180 degrees and return a value of −1 if the angle is 180 degrees or larger. We may use the `if` *conditional* statement to check the angle and return either −1 or +1 depending on the angle's value.

Since the angle value in this program actually extends beyond 360 degrees, our revised `wave` function will have to re-interpret any angle value greater than 360 as an equivalent angle between 0 and 360 degrees. This calculation is surprisingly easy to do in C/C++, using the *modulus* operator (`%`). This arithmetic operation returns the "remainder" value of an integer quotient. If we take the given angle value and use modulus to divide by 360 and return the remainder, the result will be a periodic sequence from 0 to 360 and back to 0 again, even as the actual angle value grows beyond 360.

One caveat to the use of modulus is that it only functions with integer values, and here the argument to the `wave` function is a floating-point variable. One solution to this mis-match of variable types is to *cast* the floating-point variable `degrees` into integer form by writing `(int)(degrees)`. The modified `wave` function in its entirety appears below:

```
float wave (float degrees)
{
  if ((int)(degrees) % 360 < 180.0)
    return 1.0;

  if ((int)(degrees) % 360 >= 180.0)
    return -1.0;
}
```

The result of this simulation, still using a simulated time period of 5 seconds, is as follows:

```
Total energy dissipated = 50005 Joules
DC equivalent power = 10001 Watts
DC equivalent voltage = 100.005 Volts (RMS)
```

Here we see how the peak value of a square-wave electrical voltage (or current) is equal to its RMS value. Unlike a sinusoid where the magnitude varies up and down over time, a square wave is either 100% positive or 100% negative at all times. Thus, a square wave "works just as hard" as DC, making its peak and RMS values equal.

That was so much fun, let's do it again! This time we will simulate a *sawtooth* waveform that builds linearly from −1 to +1 as the angle progresses from 0 to 360 degrees, then drops back down to −1 to repeat the pattern. Once again we will need to use the modulus function of the `degrees` angle value compared to 360 to have a periodic sequence of values bound between 0 and 360, but this time we will not have to use any `if` statements to conditionally return different values:

```
float wave (float degrees)
{
  return (float)(((int)(degrees) % 360) - 180) / 180;
}
```

Following the order of operations enforced by the parentheses, this single line of code first casts `degrees` into integer format to prepare it for the modulus operation, then it executes modulus against a value of 360, then it subtracts 180 to make the number value span from −180 to +180 (i.e. a true AC pattern that goes negative and positive), then re-casts that value into floating-point format, and finally divides by 180 to limit its range from −1 to +1.

The result of this simulated 100 Volt (peak) sawtooth wave over the same time period (5 seconds) as before, is as follows:

```
Total energy dissipated = 16668.4 Joules
DC equivalent power = 3333.68 Watts
DC equivalent voltage = 57.738 Volts (RMS)
```

According to the simulation, an AC sawtooth source of 100 Volts and 60 Hertz powering a 1 Ohm load resistance for 5 seconds will expend approximately 16668 Joules of energy, equivalent to a 57.738 Volt DC source powering the same size load resistance for the same amount of time. The RMS figure of 57.738 Volts agrees with the theoretical peak-to-RMS conversion factor of $\frac{\sqrt{3}}{3}$ used for triangular wave-shapes.

5.6 Using C++ to compute RMS and Average values

While some digital multimeters have true *RMS* measurement capability, all analog multimeters are inherently *average*[15]-responding. Since RMS measurements are generally desirable for electrical power systems, and AC power systems generally exhibit sinusoidal wave-shapes, analog multimeters are calibrated such that their inherent average-response to the AC voltage or current will be intentionally skewed to represent the RMS value of that sinusoidal quantity.

It is informative to be able to compare the RMS and average values of a wave-shape given its peak value, and we may use C++ to do this. Computing the RMS value of a waveform using C++ is shown in detail in section 5.5 beginning on page 62.

For this programming exercise we will borrow most of the code from the RMS-only algorithm previously referenced, eliminating unnecessary lines and adding in a few more to compute the average value as well. This program will output two lines of text, one describing the RMS value in relation to peak value, and the other describing average value in relation to peak

[15] The average value of an AC waveform centered around zero with a 50% duty cycle will, of course be zero. When we say "average-responding" what we really mean is that the AC instrument responds to the average of all the *absolute values* of the AC waveform. Another way of thinking about this is to imagine the analog instrument averaging a *full-wave rectified* version of the AC waveform where all instantaneous values are positive.

Shown here is the listing of a C++ program performing all these calculations. Values of AC peak voltage, resistance, time period, and time step interval are embedded in the code, so that when it is compiled and run it reports equivalent RMS and average voltages:

```
#include <iostream>
#include <cmath>

using namespace std;

float wave (float);

int main (void)
{
  float t, v, dcv;
  float t_int = 0.0001;
  float t_end = 1.0;
  float freq = 60.0;
  float vpk = 100.0;
  float e = 0.0;
  float avg = 0.0;
  float r = 1.0;

  for (t = 0; t <= t_end; t = t + t_int)
    {
      v = vpk * wave (360 * freq * t);
      e = e + (pow (v, 2) / r * t_int);
      avg = avg + (abs(v) * t_int);
    }

  dcv = sqrt (e / t_end * r);
  avg = avg / t_end;

  cout << "RMS = " << dcv / vpk * 100 << "\% of peak"<< endl;
  cout << "Average = " << avg / vpk * 100 << "\% of peak"<< endl;

  return 0;
}

float wave (float degrees)
{
  float radians;
  radians = degrees * M_PI / 180.0;
  return sin (radians);
}
```

When we run this program we get the following results:

```
RMS = 70.7088% of peak
Average = 63.6584% of peak
```

According to the simulation, a sinusoidal waveform has an RMS value equal to 70.7088% of its peak value and an average value equal to 63.6584% of its peak value.

Next we will replace the `wave` function's sinusoidal code with code simulating a square wave (shown below) and re-run the program (output shown below that):

```
float wave (float degrees)
{
  if ((int)(degrees) % 360 < 180.0)
    return 1.0;

  if ((int)(degrees) % 360 >= 180.0)
    return -1.0;
}
```

```
RMS = 100% of peak
Average = 100.003% of peak
```

Here we see how both the RMS and average values of a square wave are equal to its peak value.

Lastly, we will simulate a sawtooth waveform and repeat the test:

```
float wave (float degrees)
{
  return (float)(((int)(degrees) % 360) - 180) / 180;
}
```

```
RMS = 57.7386% of peak
Average = 50.0026% of peak
```

Here we see the RMS value of a sawtooth wave is 57.7386% of its peak value, while the average is one-half its peak.

In addition to RMS, average, and peak (*crest*) values for AC waveforms, there are ratios expressing the proportionality between some of these fundamental measurements. The *crest factor* of an AC waveform, for instance, is the ratio of its peak value divided by its RMS value. The *form factor* of an AC waveform is the ratio of its peak value divided by its average value.

Our C++ program may be modified to display both of these ratios quite easily by adding two more `cout` lines, a partial code listing shown here:

```
cout << "Crest factor = " << vpk / dcv << endl;
cout << "Form factor = " << vpk / avg << endl;
```

Results for sine waves, square waves, and triangle waves are summarized here, respectively, from three runs of this program (each with a different `wave` function):

Sine wave

```
RMS = 70.7088% of peak
Average = 63.6584% of peak
Crest factor = 1.41425
Form factor = 1.57089
```

Square wave

```
RMS = 100% of peak
Average = 100.003% of peak
Crest factor = 1
Form factor = 0.99997
```

Sawtooth wave

```
RMS = 57.7386% of peak
Average = 50.0026% of peak
Crest factor = 1.73194
Form factor = 1.9999
```

Theoretical crest- and form-factor ratios are $\sqrt{2}$ and $\frac{\pi}{2}$ for sine waves; 1 and 1 for square waves; $\sqrt{3}$ and 2 for sawtooth and triangle waves.

As mentioned at the beginning of this section, analog voltmeters and ammeters naturally respond to the average value of whatever AC quantity being measured, not the RMS value. Since RMS values are widely used in electrical power circuit measurements, analog instrument calibrations are intentionally "skewed" to represent the RMS value rather than the average, assuming a sinusoidal wave-shape. This "skewing" takes the form of artificially *multiplying* the analog instrument's natural response so that it registers more voltage or current than it naturally would, knowing that for a sine wave the RMS value is more than the average value[16].

Measurement errors arise when any analog instrument calibrated to register sinusoidal-RMS values senses a non-sinusoidal waveshape, this error related to the ratio of crest to form factors (i.e. the ratio of RMS to average value for that waveshape). A properly-calibrated AC analog voltmeter or ammeter thus has an intentional calibration factor of $\frac{\pi}{2\sqrt{2}}$ (approximately 1.111) to make their average-responding movements register the equivalent RMS value for a sine wave. In order to compute the measurement error of an analog meter, we simply need to compare this ratio against the actual average-to-RMS ratio of the waveform being measured. Once again, this is relatively easy to do in C++, adding just one[17] more `cout` line to the program.

To express error as a percentage deviation from some ideal value, we use the following formula:

$$\text{Error} = \frac{\text{Actual} - \text{Ideal}}{\text{Ideal}} \times 100\%$$

For our purposes in computing analog meter measurement error, the "actual" ratio of RMS to average for the tested waveform will be coded as `dcv / avg` while the "ideal" is $\frac{\pi}{2\sqrt{2}}$. Our added code is shown below:

```
cout << "Analog meter error = " <<
((dcv / avg) - (M_PI / (2 * sqrt(2)))) / (M_PI / (2 * sqrt(2))) * 100 <<
"\%" << endl;
```

[16] It can become very confusing trying to figure out which way an analog instrument's calibration needs to be skewed in order to register RMS rather than average Volts or Amperes. A helpful problem-solving strategy is to put numerical values to the problem, imagining a sinusoidal AC voltage with a peak value of 100 Volts energizing both RMS-indicating and average-indicating meters. The RMS-indicating meter would read 70.71 Volts while the average-indicating meter would read 63.66 Volts. Clearly, the average-indicating meter would have to have its reading *multiplied* to match the RMS-indicating meter, and so this is a way to confidently conclude the need for a *multiplying* factor (rather than a *dividing* factor) in the analog instrument to force it to register RMS values for a sinusoidal voltage or current.

[17] This is a rather long line of code, so it is shown here split into three lines on the page. This is permissible in C and C++ because these languages do not respect whitespace, instead relying on the semicolon character (`;`) to denote the end of the code line.

Re-running this program three more times (one for each waveshape) yields the following results:

Sine wave

```
RMS = 70.7088% of peak
Average = 63.6584% of peak
Crest factor = 1.41425
Form factor = 1.57089
Analog meter error = 0.00293172%
```

Square wave

```
RMS = 100% of peak
Average = 100.003% of peak
Crest factor = 1
Form factor = 0.99997
Analog meter error = -9.97102%
```

Sawtooth wave

```
RMS = 57.7386% of peak
Average = 50.0026% of peak
Crest factor = 1.73194
Form factor = 1.9999
Analog meter error = 3.96073%
```

As you can see here, our sinusoidal simulation results in a measurement error of practically zero while the square wave simulation makes the analog meter register -9.97102% too low and the sawtooth wave simulation makes the analog meter register 3.396073% too high.

Chapter 6

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?

6.1 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.

6.1.1 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.

6.1.2 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.

Energy

Conservation of Energy

Voltage

Electrical source

Electrical load

Alternating current

Direct current

Electromagnetism

Electromagnetic induction

Faraday's Law of electromagnetic induction

Mutual induction

Wave

Ohm's Law

Joule's Law

Kirchhoff's Voltage Law

Kirchhoff's Current Law

Period

Frequency

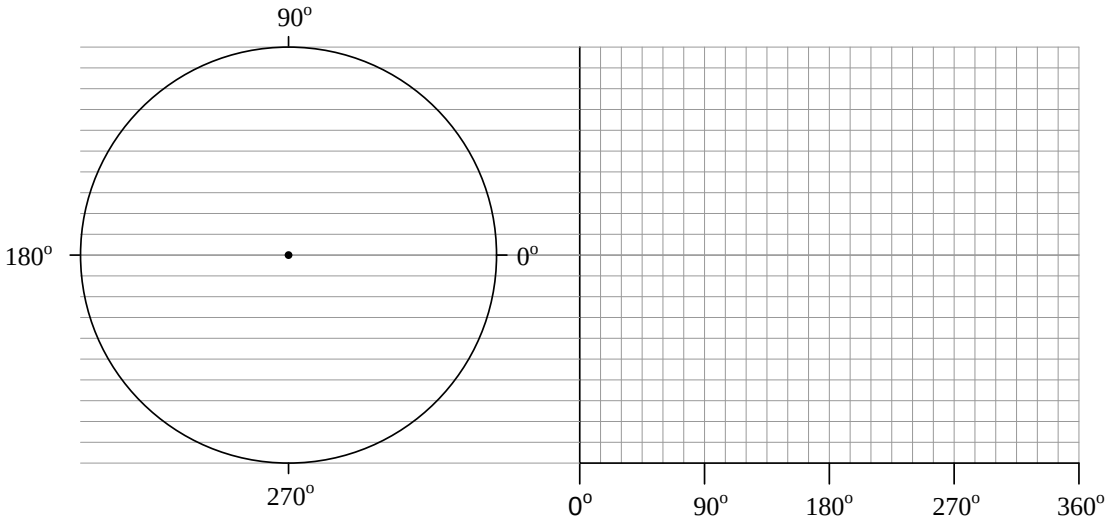
Wavelength

RMS

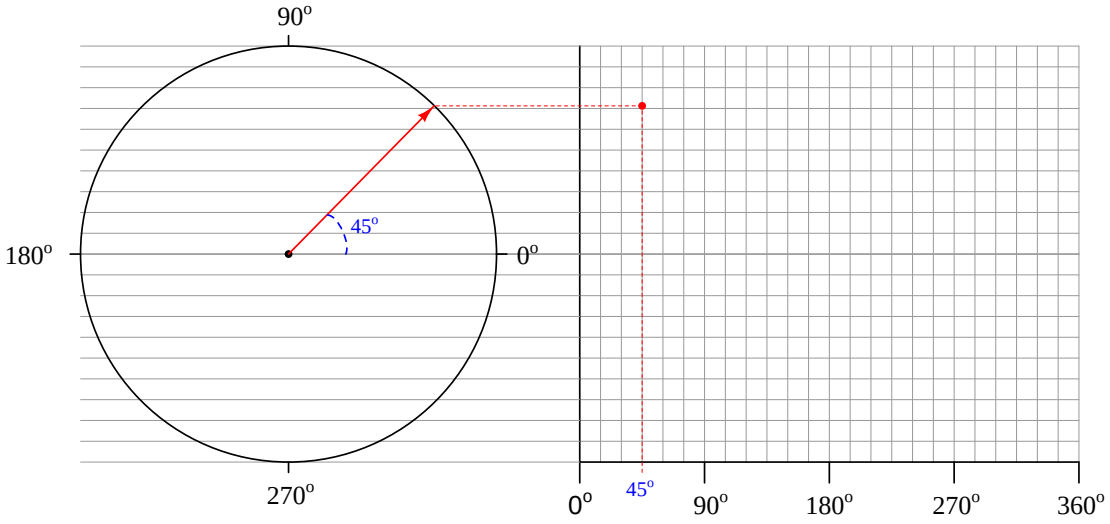
Phase

6.1.3 Plotting a sine wave

Alternating current produced by electromechanical generators (or *alternators* as they are sometimes designated) typically follows a sine-wave pattern over time. Plot a sine wave on the following graph, by tracing the height of a rotating vector inside the circle to the left of the graph:



To illustrate the principle here, I will show how the point is plotted for a rotation of 45^o:



You may wish to use a *protractor* to precisely mark the angles along the rotation of the circle, in making your sine-wave plot.

Challenges

- Explain how you could similarly plot a *cosine* wave.

6.1.4 Wire and insulation sizing

In calculating the size of wire necessary (i.e. the conductor's "gauge" value) to carry sinusoidal alternating current to a high-power load, which type of measurement is the best to use for current: peak, average, or RMS? Explain why.

In calculating the thickness of wire insulation necessary to withstand sinusoidal alternating voltage in a high-voltage power system, which type of measurement is the best to use for voltage: peak, average, or RMS? Explain why.

Challenges

- Suppose we are selecting a rectifying diode to use in an AC-DC converter circuit. Which parameter of the diode best relates to RMS, versus to peak, of the AC quantities being rectified?
- Suppose the AC frequency was extremely slow (e.g. 1 cycle per *hour*) rather than 60 Hz as is typical for US power systems. Would this low frequency assumption change your answers to these questions?

6.2 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.

6.2.1 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.

6.2.2 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	

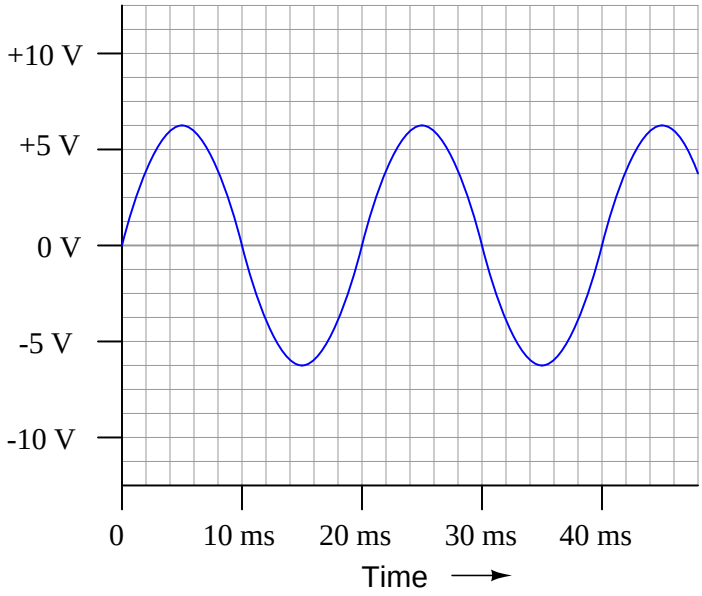
Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference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

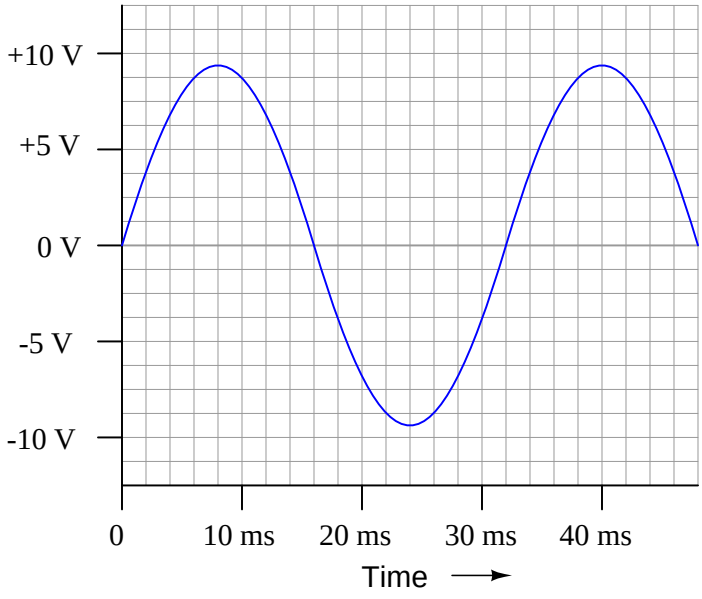
6.2.3 AC voltage oscillographs

Quantify the following parameters in this oscillograph of an AC voltage:



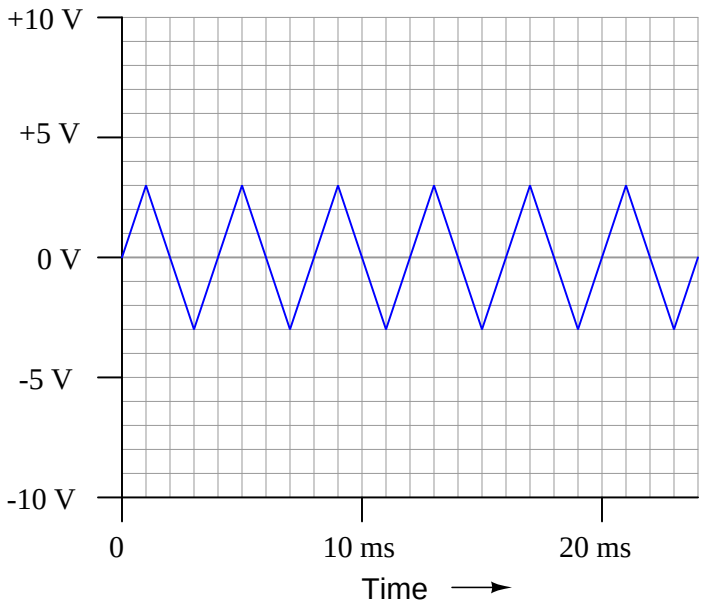
- Period
- Frequency
- Amplitude

Quantify the following parameters in this next oscillograph of an AC voltage:



- Period
- Frequency
- Amplitude

Quantify the following parameters in this next oscillograph of an AC voltage:



- Period
- Frequency
- Amplitude

Challenges

- How would any of these oscillographs' appearances change if the period increased?
- How would any of these oscillographs' appearance change if the frequency increased?
- How would any of these oscillographs' appearance change if the amplitude increased?
- Calculate the RMS value for each of these voltage waveforms.

6.2.4 Radio wavelength

Radio waves are comprised of oscillating electric and magnetic fields, which radiate away from sources of high-frequency AC at (nearly) the speed of light. An important measure of a radio wave is its *wavelength*, defined as the distance the wave travels in one complete cycle.

Suppose a radio transmitter operates at a fixed frequency of 950 kHz. Calculate the approximate wavelength (λ) of the radio waves emanating from the transmitter tower, in the metric distance unit of meters. Also, write the equation you used to solve for λ.

Challenges

- A very common type of radio antenna is called a *quarter-wave* design, so named because of its physical length compared to the wavelength of the radio waves emitted and/or received by it. How long would this antenna need to be for this particular transmitter?

6.2.5 Load power at 50 Volts

Suppose a DC power source with a voltage of 50 Volts is connected to a 10 Ω load. How much power will this load dissipate?

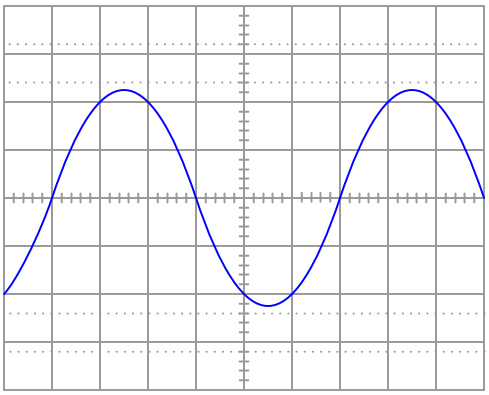
Now suppose the same 10 Ω load is connected to a sinusoidal AC power source with a *peak* voltage of 50 Volts. Will the load dissipate the same amount of power, more power, or less power? Explain your answer.

Challenges

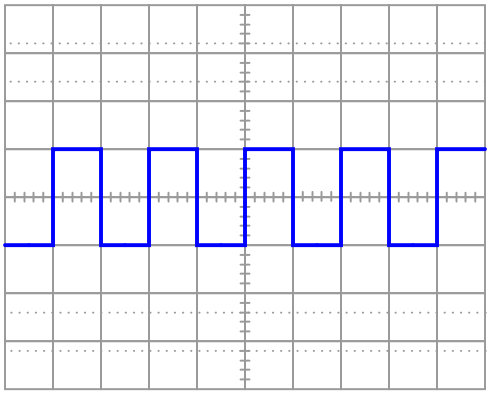
- Does the shape of the AC voltage's wave matter?

6.2.6 RMS values from oscillographs

Determine the RMS amplitude of this sinusoidal waveform, as displayed by an oscilloscope with a vertical sensitivity of 0.2 Volts per division:



Now determine the RMS voltage of this square-wave signal, as displayed by an oscilloscope with a vertical sensitivity of 0.5 Volts per division:

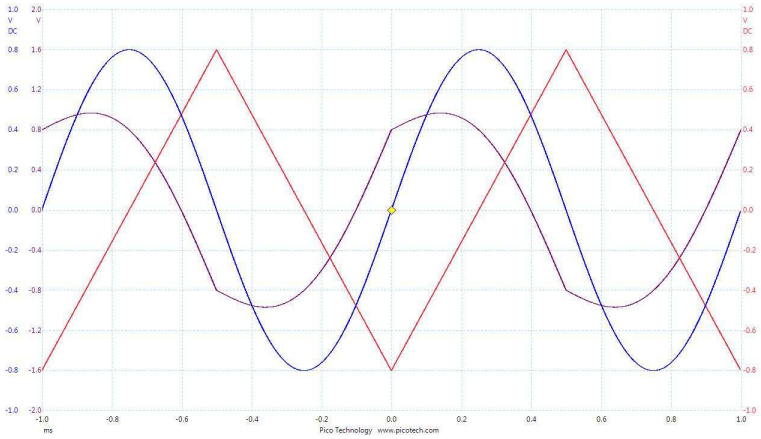


Challenges

- Suppose after calculating the RMS values of these voltage signals, you subsequently discover the oscilloscope's probe was set for a 1:10 division ratio when you thought it was 1:1. How would this affect your calculations, if at all?
- Explain why the timebase setting of the oscilloscope is irrelevant to the question of determining RMS value.

6.2.7 Phase shift from oscillographs

Identify the amount of phase shift between the sine and triangle waves, and also which one is *leading* versus *lagging*:

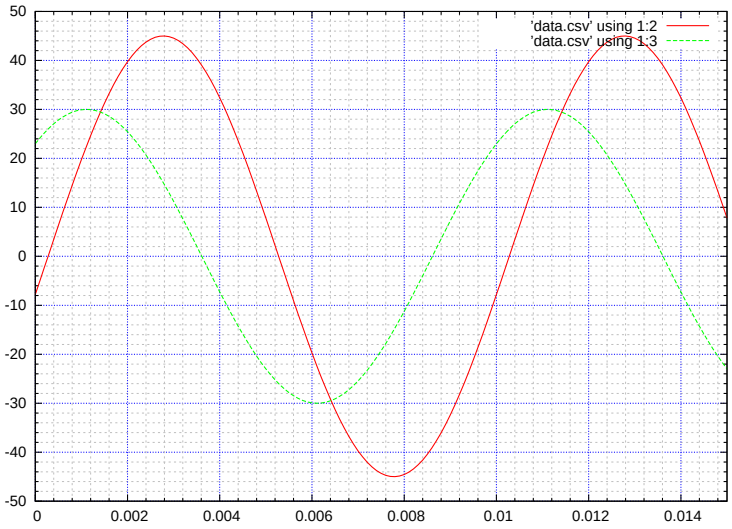


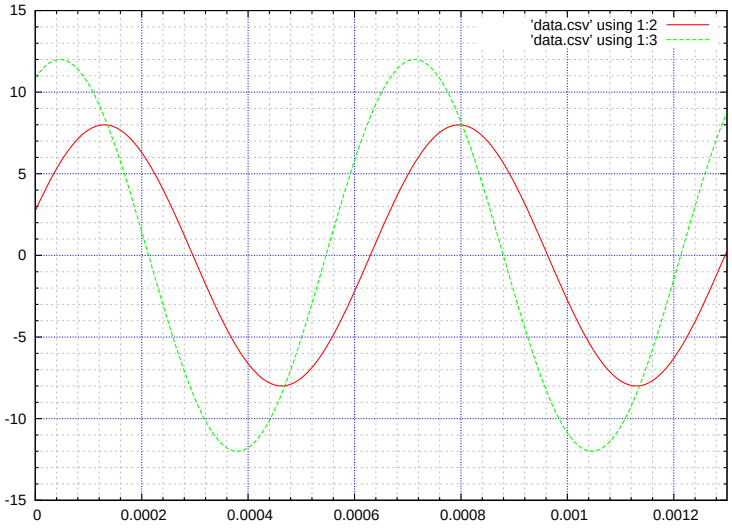
Challenges

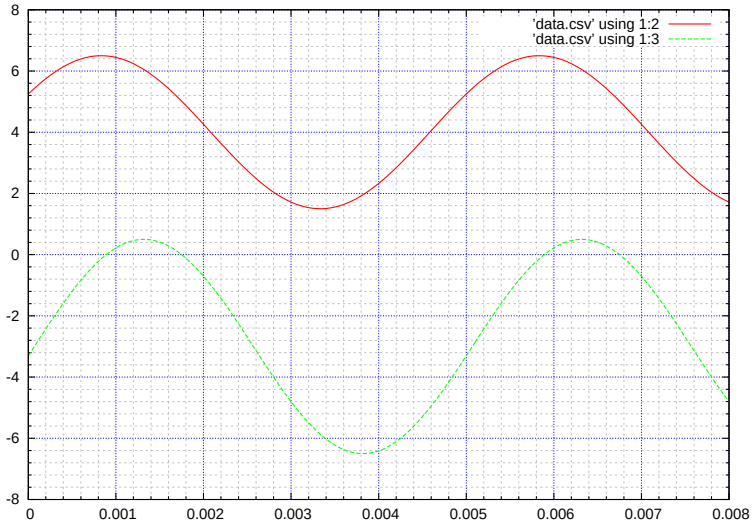
- Suppose after calculating the RMS values of these voltage signals, you subsequently discover the oscilloscope's probe was set for a 1:10 division ratio when you thought it was 1:1. How would this affect your calculations, if at all?
- Explain why the timebase setting of the oscilloscope is irrelevant to the question of determining phase shift.

6.2.8 Oscillographs comparing sinusoids

Examine the following oscillographs, and from them determine the amplitudes of and phase shift separating the two sinusoids, as well as their frequency and period. In each case the vertical axis has a unit of *Volts* and the horizontal axis a unit of *seconds*:







Challenges

- How can we tell both waveforms in each oscillograph have the same frequency, and why might this fact be important to us?

6.2.9 D'Arsonval versus true-RMS meter measurements

Suppose two voltmeters are connected to source of "mains" AC power in a residence, one meter is analog (using a *D'Arsonval*-style permanent magnet moving coil meter mechanism) while the other is true-RMS digital meter. Both meters register 117 Volts while connected to the AC mains.

Suddenly, a large electrical load is turned on somewhere in the system, both reducing the mains voltage and slightly distorting the shape of its waveform. The overall effect of this is average AC voltage has decreased by 4.5% from where it was, while RMS AC voltage has decreased by 6% from where it was. How much voltage does each voltmeter register now?

Challenges

- Calculate the peak value of the undistorted 117 Volt AC waveform.

6.2.10 VOM versus DMM voltage measurements

Two voltmeters are connected in parallel to a signal generator, simultaneously measuring the voltage output by that AC source. One meter is a Fluke model 87-III true RMS digital multimeter (DMM), while the other is a Simpson model 260 volt-ohm-milliammeter (VOM). These two meters are shown measuring three different voltage waveforms output by the signal generator: *sine wave*, *square wave*, and *triangle* wave. In each case, the signal generator's output amplitude has been adjusted to render a measurement of 4 Volts on the VOM (set to the 10-Volt scale), while the DMM shows the RMS value of that same voltage.

Identify which of these photographs was taken while measuring the sine-wave voltage, the square-wave voltage, and the triangle-wave voltage:

Challenges

- Sketch superimposed square, sine, and triangle waves all having the same peak values. Compare these three wave-shapes and then comment on how their relative *areas* (i.e. the "integration" over time, graphically equivalent to the area bounded by each wave and the zero line) relate to a VOM's measurement error.

6.3 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.

6.3.1 Detecting AC mains distortion

The ideal wave-shape for AC voltages and currents in an electric power system is sinusoidal, because that is what rotating generators naturally output. However, the presence of certain types of "switching" loads in AC power systems tends to distort the naturally pure sinusoidal shape of the system's voltages and currents. These distortions may actually be harmful to certain system components, a topic too complex to discuss here.

Explain how we may use a pair of voltmeters – one being a true-RMS digital voltmeter and the other being a high-quality analog voltmeter – to tell whether or not the AC "mains" voltage available at a power receptacle is distorted or not.

Challenges

- Identify a single instrument that would be able to answer this question for us, rather than two voltmeters of different 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.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

Gardner, Carl E., *Essentials of Music Theory*, Carl Fischer Inc., Cooper Square, New York, 1912.

Dineen, Joe and Bridges, Mark, *The Gig Bag Book of Theory & Harmony*, Amsco Publications, Music Sales Corporation, New York, New York, 2000.

Tapper, Thomas, *First Year Musical Theory*, Arthur P. Schmidt, New York, New York, 1912.

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.

25 August 2024 – added another section to the Introduction chapter specifically for challenges related to this module's topic.

1 July 2024 – divided the Introduction chapter into two sections, one for students and one for instructors, and added content to the instructor section recommending learning outcomes and measures.

26 January 2024 – simplified the RMS/peak ratios described in the Tutorial.

25 July 2023 – added a Case Tutorial chapter with a section on building and using a sensitive audio detector.

3 February 2023 – replaced all instances of "magnitude" with "amplitude" for better clarity.

25 January 2023 – corrected an error in image_4930 where the phase shift time was incorrectly labeled. Also added some instructor notes.

28 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.

26 July 2022 – added "primary" and "secondary" labels to more transformer diagrams for clarity.

14 June 2022 – minor edits to the Tutorial section on quantifying AC, as well as to the Derivations and Technical References section on musical pitch.

18 May 2022 – minor formatting changes made to the Tutorial chapter.

22 March 2022 – divided Tutorial into sections.

8 May 2021 – commented out or deleted empty chapters.

18 April 2021 – changed all lower-case Greek letter "phi" symbols (ϕ) to upper-case (Φ).

5 February 2021 – added brief reference to transformers in the Tutorial as well as in the Introduction.

27 January 2021 – minor clarifying edits to illustrations of wave parameters in the Tutorial, as well as a new image added to the Tutorial showing t_{shift} and t_{period} clearly on the oscillograph.

28 October 2020 – minor additions to the Introduction chapter, and also to the "Wire and insulation sizing".

5 October 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 minor edits to the Tutorial.

20 April 2020 – added a another Programming Reference section using C++ to plot two sinusoidal waveforms with their own phase shifts. Then, used this program to generate some oscillographs for more Quantitative Reasoning questions.

22 March 2020 – added a Quantitative Reasoning problem comparing the displays of a DMM and VOM for sine waves, square waves, and triangle waves. Also added another Programming Reference section using C++ to simulate both the RMS and Average values of various AC wave-shapes, with a discussion on analog instrument calibration and the errors encountered when measuring non-sinusoidal waveforms.

9 March 2020 – complete Foundational Concepts section, and added discussion of phase into the Tutorial.

8 March 2020 – document first created.

Index