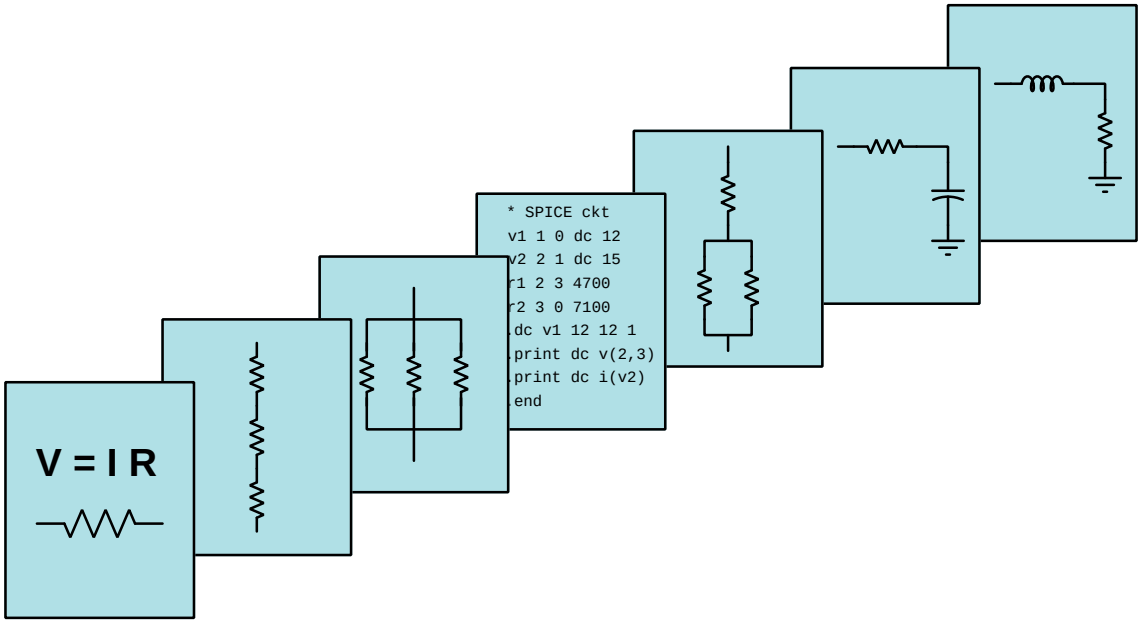
Modular Electronics Learning (ModEL) project



Phasor Mathematics

© 2017-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 30 August 2026

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit `http://creativecommons.org/licenses/by/4.0/` or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.

Contents

1 Introduction **3**
- 1.1 Recommendations for students . . . 3
- 1.2 Challenging concepts related to phasor mathematics . . . 5
- 1.3 Recommendations for instructors . . . 7

2 Simplified Tutorial **9**

3 Full Tutorial **17**
- 3.1 AC versus DC . . . 18
- 3.2 Additive quantities . . . 21
- 3.3 Complex numbers and exponential functions . . . 29
- 3.4 Trigonometric conversions . . . 35
- 3.5 Summary . . . 37

4 Derivations and Technical References **39**
- 4.1 Derivation of Euler's Relation . . . 40
- 4.2 Complex-number arithmetic . . . 45
  - 4.2.1 Negating complex numbers . . . 46
  - 4.2.2 Adding complex numbers . . . 46
  - 4.2.3 Subtracting complex numbers . . . 46
  - 4.2.4 Multiplying complex numbers . . . 47
  - 4.2.5 Dividing complex numbers . . . 47
  - 4.2.6 Reciprocating complex numbers . . . 48
  - 4.2.7 Calculator tips . . . 48

5 Programming References **49**
- 5.1 Programming in C++ . . . 50
- 5.2 Programming in Python . . . 54
- 5.3 Simple plotting of sinusoidal waves using C++ . . . 59
- 5.4 Plotting two sinusoidal waves with phase angles using C++ . . . 74
- 5.5 Simple plotting of arbitrary waveforms using C++ . . . 78

6 Questions **85**
6.1 Conceptual reasoning . . . 89
6.1.1 Reading outline and reflections . . . 90
6.1.2 Foundational concepts . . . 91
6.1.3 Radians . . . 92
6.2 Quantitative reasoning . . . 93
6.2.1 Miscellaneous physical constants . . . 94
6.2.2 Practice: complex number calculations . . . 95
6.2.3 Adding sine waves . . . 99
6.2.4 Hand-plotting a sinusoidal sum . . . 101
6.2.5 Phasor addition of two AC voltages . . . 103
6.2.6 Simple phasor diagrams . . . 104
6.2.7 Series AC voltages . . . 105
6.2.8 Parallel AC currents . . . 107
6.3 Diagnostic reasoning . . . 108
6.3.1 Incorrect voltage calculation . . . 109

A Problem-Solving Strategies **111**

B Instructional philosophy **113**
B.1 First principles of learning . . . 114
B.2 Proven strategies for instructors . . . 115
B.3 Proven strategies for students . . . 117
B.4 Design of these learning modules . . . 118

C Tools used **121**

D Creative Commons License **125**

E References **133**

F Version history **135**

Index **137**

Chapter 1

Introduction

1.1 Recommendations for students

A very useful tool for the simplification of AC circuit analysis is the use of *complex numbers* to represent voltages (V), currents (I), and impedances (Z). Any such quantity represented as a complex number is called a *phasor* because that complex-number quantity represents both the magnitude as well as the phase angle of that parameter. If we represent all AC circuit quantities as phasors, we find that many of the principles and laws familiar from our study of DC circuit analysis such as Ohm's and Kirchhoff's Laws still hold true. The only challenge remaining to AC circuit analysis once we embrace the notion of using phasors is the task of performing arithmetic operations such as addition, subtraction, multiplication, and division on complex numbers rather than the more customary "real" numbers used in DC circuit analysis. This module explores the use of complex-number quantities to unify AC circuit analysis with DC circuit analysis.

Important concepts related to phasors include **electromagnetic induction**, mathematical **sine** and **cosine** functions, **polar** versus **rectangular** notation of complex numbers, **real** versus **imaginary** quantities, **Kirchhoff's Voltage Law**, instantaneous **addition of waveforms**, the **Pythagorean Theorem**, **magnitude**, **frequency**, **right triangles**, **Euler's Relation**, the j operator, **angular velocity**, **exponential** functions, **adjacent** versus **opposite** versus **hypotenuse** sides of a right triangle, and **phase angle**.

This module utilizes a lot of *projections* to relate circular motion to time-based wave patterns, showing that the two are really just different ways of representing the same fundamental phenomena. You are encouraged to try sketching your own projections in order to grasp what they represent.

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

- How might an experiment be designed and conducted to mathematically demonstrate the addition of two sinusoids that are phase-shifted from one another, using a computer to perform the arithmetic? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What are some practical applications of phasors?

- What does it mean to say that an AC quantity has a *frequency*?
- How do we express the magnitude an AC quantity?
- How does the magnitude an AC quantity relate to its instantaneous amplitude (i.e. its voltage or current value at a particular instant in time)?
- From where does the shape of a sine wave or cosine wave originate?
- What means do we possess to express the magnitude of an angle?
- What practical benefit do complex numbers lend to the analysis of AC circuits?
- Why is the phase shift separating two waveforms necessary to know in order to compute their sum?
- What do the "real" and "imaginary" portions of a rectangular-format complex number represent for an AC quantity?
- What do the "magnitude" and "angle" portions of a polar-format complex number represent for an AC quantity?
- How does the construction of an AC electrical generator relate to the real and imaginary portions of an AC voltage?
- What trigonometric functions are used to convert a polar-form complex number into a rectangular-form complex number?
- What trigonometric functions are used to convert a rectangular-form complex number into a polar-form complex number?
- Which form of complex number relates most closely with the indication provided by an electrical meter, rectangular or polar?

1.2 Challenging concepts related to phasor mathematics

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.

- **Phasors representing AC amplitudes and phase shifts** – a powerful tool used for understanding the operation of AC circuits is the *phasor diagram*, consisting of arrows pointing in different directions: the length of each arrow representing the amplitude of some AC quantity (voltage, current, or impedance), and the angle of each arrow representing the shift in phase relative to the other arrows. By representing each AC quantity thusly, we may more easily calculate their relationships to one another, with the phasors showing us how to apply trigonometry (Pythagorean Theorem, sine, cosine, and tangent functions) to the various calculations. An analytical parallel to the graphic tool of phasor diagrams is *complex numbers*, where we represent each phasor (arrow) by a pair of numbers: either a magnitude and angle (polar notation), or by "real" and "imaginary" magnitudes (rectangular notation). Where phasor diagrams are helpful is in applications where their respective AC quantities *add*: the resultant of two or more phasors stacked tip-to-tail being the mathematical sum of the phasors. Complex numbers, on the other hand, may be added, subtracted, multiplied, and divided; the last two operations being difficult to graphically represent with arrows.

- **Complex numbers in calculators** – while the ability of certain scientific calculators to perform complex-number arithmetic is an enormously helpful tool for students first learning to analyze AC circuits, some of these calculators prove to be finicky in their handling and entry

of these quantities. Advice proven to be sound for all complex-number calculators is to save each and every complex-valued quantity into a memory location and then perform arithmetic operations on those stored variables rather than enter the complex numbers directly into the computation. For example, storing `3 - j4` into memory location `A` and `25 ∠ 30°` into memory location `B`, then multiplying `A × B` rather than entering `3 - j4 × 25 ∠ 30°`. Storing values into calculator memory and then retrieving them as needed for calculations is actually sound advice for many reasons, but many students resist taking these "extra" steps and as a result incur all the risks of hand-entering values (e.g. rounding errors due to truncating, keystroke errors when the same value must be used more than once, crowded displays where you cannot see the whole calculation, order-of-operations errors when complex numbers aren't enclosed in parentheses, etc.). Complex-number calculations reward good practices through consistently good results!

- **Complex numbers in measurement** – complex numbers may be expressed in either *rectangular* or *polar* form, either one of these being perfectly valid. However, measurement instruments such as multimeters only provide the magnitude of the polar form of the voltage or current in question. For example, if a component's voltage is 4.8 Volts RMS $\angle$ 35^o with the circuit's source voltage being the phase reference (0^o), an voltmeter reading that voltage will simply register 4.8 Volts RMS. An oscilloscope simultaneously measuring that component voltage on one channel and the source voltage on another will show the 35^o shift on the horizontal axis between the two waveforms.

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.

  Assessment – Students present their "Conceptual Toolbox" Theory Journal entries listing important concepts applied in the Tutorial text(s). A "Conceptual Toolbox" serves as inspiration when solving problems, each concept suggesting a potential problem-solving action. Challenge students to collaboratively generate a Conceptual Toolbox for the day's topic during the class session and then use it when solving problems together, thus fostering the habit of reasoning from principles rather than recalling procedures!

- **Outcome** – Apply the concept of phasor addition

  Assessment – plot the waveform representing the sum of two sinusoidal waves that are not in-phase with each other; e.g. pose problems in the form of the "Hand-plotting a sinusoidal sum" Quantitative Reasoning question.

  Assessment – interpret the phasor-diagram sums of two sinusoidal waves; e.g. pose problems in the form of the "Phasor addition of two AC voltages" Quantitative Reasoning question.

  Assessment – calculate additive AC quantities (e.g. series voltages, parallel currents) using complex-numbered values; e.g. pose problems in the form of the "Series AC voltages" and/or "Parallel AC currents" Quantitative Reasoning questions.

- **Outcome** – Apply the concept of complex numbers to phasor diagrams

  Assessment – write polar- and rectangular-form complex numbers for given vectors sketched on a phasor diagram; e.g. pose problems in the form of the "Simple phasor diagrams" Quantitative Reasoning question.

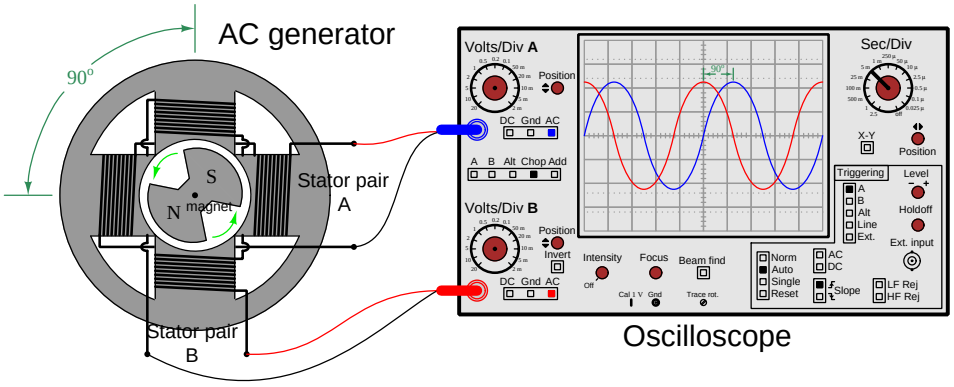
- **Outcome** – Independent research

  Assessment – Read and summarize in your own words reliable historical documents on the subject of applying complex numbers to AC circuit calculations. Recommended readings include books written by Charles Proteus Steinmetz.

Chapter 2

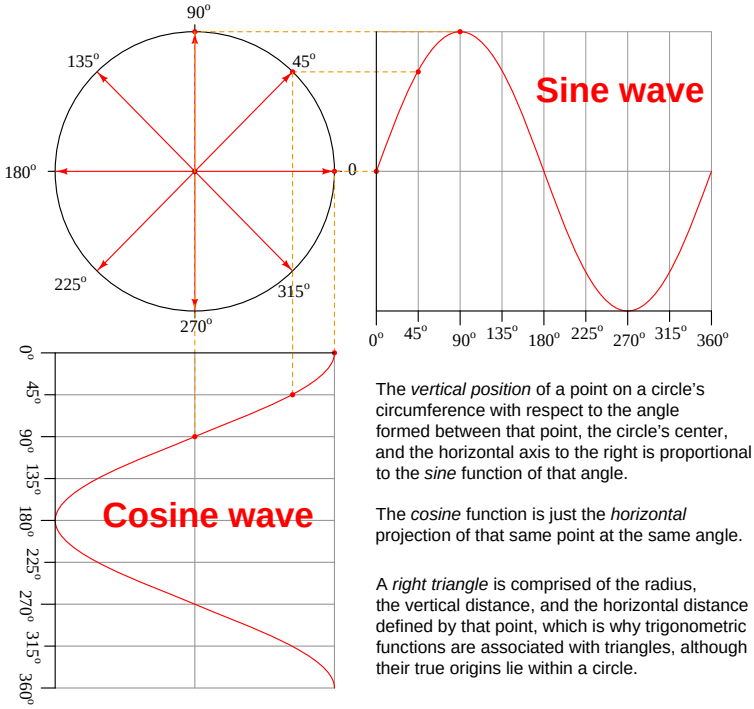
Simplified Tutorial

Spinning a magnetized rotor past sets of stationary "stator" coils (also called *stator windings*) will produce alternating (AC) voltages resembling sine waves when displayed on an oscilloscope. This is the basis of all electromagnetic *generators*:



If multiple pairs of stator windings exist at different positions around the rotor's circumference, the AC voltages generated by each winding pair will be shifted in phase from one another because the rotor's poles can never align with all stator windings simultaneously: each stator winding reaches its "peak" voltage at a different rotor position. For a simple two-pole generator such as the one shown, the angle separating those winding positions will be the exact same phase-shift angle separating their AC voltage waveforms. In this example, that shift is 90^o.

The sinusoidal[1] shape of these voltage waveforms over time is no coincidence. Rather, it is a geometric fact proven by plotting the horizontal and vertical projections of a point moving around the circumference of a circle:



This geometry gives us ways to fully represent quantities within AC circuits. DC circuits pose little challenge in this way because the quantities are largely unchanging, but with AC circuits we have voltages and currents that continually vary over time, as well as other quantities such as *impedance* where the phase shift between voltage and current must be known. For example, a battery in a DC circuit may output a constant voltage of 6 Volts, but an AC voltage source with a 6 Volt "peak" rating continually cycles from zero Volts to +6 Volts to zero Volts to −6 Volts to zero Volts repeatedly. The math we use must be capable of expressing this reality.

Numerical values such as "6 Volts" are called *scalar* numbers, but such plain and simple quantities fail to capture the complexity of an AC waveform when that waveform is one of multiple waveforms in a system that happen to be phase-shifted from one another. A type of mathematical quantity called a *complex number* is better-suited for this purpose, and may be written in two different forms: *polar* and *rectangular*. The "polar" form of a complex number relates to the circle portion in the previous diagram, while the "rectangular" form relates to the projections of that circle (the sine and cosine waves).

[1] The term "sinusoid" refers to a shape resembling a sine wave.

The Swiss mathematician Leonhard Euler (1707-1783) discovered a relationship between exponential functions and trigonometric functions (cosine and sine), expressed in the following formula known as *Euler's Relation*:

$$e^{j\theta} = \cos\theta + j\sin\theta$$

Where,

e = Euler's number (approximately equal to 2.718281828)

θ = Angle of phasor, in radians

$\cos\theta$ = Horizontal projection of a unit phasor (along a real number line) at angle θ

j = Imaginary "operator" equal to $\sqrt{-1}$, alternatively represented as i

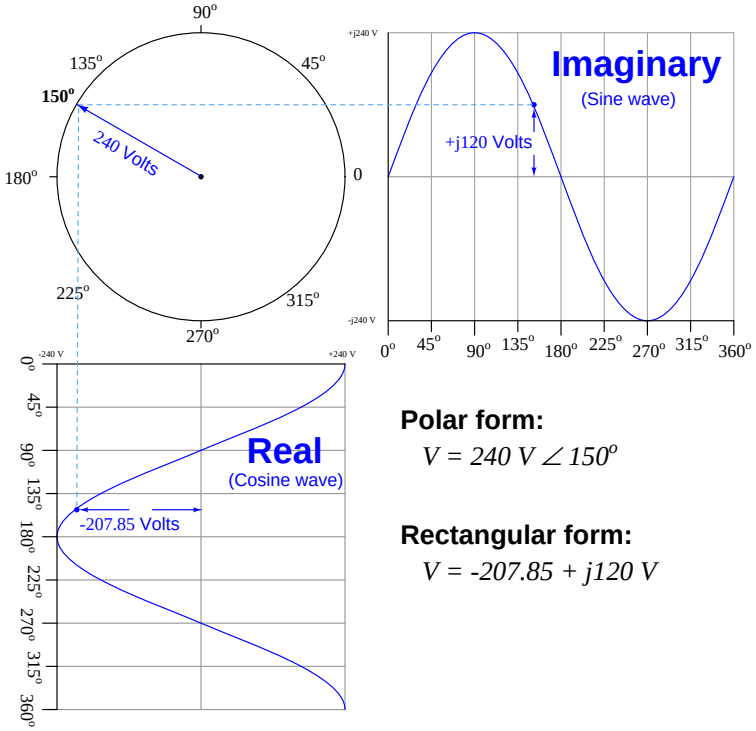
$j\sin\theta$ = Vertical projection of a unit phasor (along an imaginary number line) at angle θ

This formula makes use of the "imaginary operator", usually expressed by mathematicians as i but expressed by engineers and technicians as j so as to avoid confusing it with *electric current* (I or i). The trigonometric terms $\cos\theta$ and $\sin\theta$ represent the "real" and "imaginary" terms of a complex number in rectangular form, while the exponential term $e^{j\theta}$ represents the unit vector and its angle θ of a complex number in polar form. For any complex numbers having a polar magnitude other than one (1), a multiplier M may be included in this formula as follows:

$$Me^{j\theta} = M\cos\theta + jM\sin\theta$$

Relating this to our hypothetical AC generator with two out-of-phase stator windings, M would be the peak voltage of that generator's sinusoidal output while θ would be its shaft position at any given moment in time.

The following example should make this clear. Here we have both the polar and rectangular representations of our dual-winding AC generator's output voltage at the instant in time where its shaft position is 150^o. One winding pair has been labeled "Real" and the other "Imaginary":



This generator happens to output 240 Volts *peak*, which we can tell directly from the magnitude portion of polar notation: $240\ V \angle 150^o$. At that particular moment in time when the shaft reaches the 150^o position, the two winding pairs are outputting -207.85 Volts and $+120$ Volts, respectively. This is directly represented by the rectangular notation: $-207.85 + j120$ V. If we wished to use Euler's exponential notation for the polar notation we could write $240e^{j\frac{150\pi}{180}}$, $\frac{\pi}{180}$ being a conversion factor to turn 150 degrees into the unit of radians that Euler's Relation demands. The less mathematically rigorous shorthand notation of $240 \angle 150^o$, though, is more common among electrical practitioners.

These two mathematical forms, $240\ V \angle 150^o$ and $-207.85 + j120$ V, are just alternative ways of expressing the same thing: the voltage output by this generator at a specific moment in time. Even if our hypothetical generator only had one winding pair (making the other winding pair *truly* imaginary), there would still be merit in rectangular notation because the combination of real and imaginary terms completely define that point along the whole shape of the wave. Neither the real (-207.85 V) nor the imaginary ($+120$ V) terms by themselves uniquely[2] identify the coordinate at 150^o, but the combination of the two terms does.

[2]Examining the projected sine and cosine waves, we see there are *two* different points in time where the "real" voltage reaches -207.85 Volts and *two* different points in time where the "imaginary" voltage reaches $+120$ Volts, but these two conditions only exist simultaneously at *one* point in time (corresponding to 150^o).

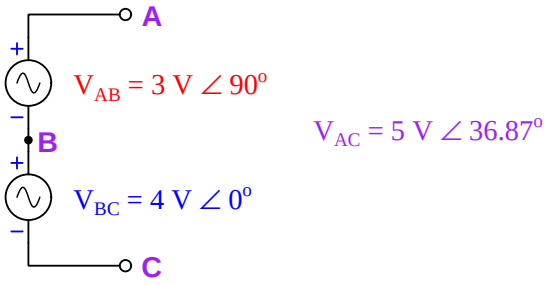
In fact, complex-number notation is useful for representing a great many quantities in AC circuits, not just generator voltage. We may represent any current, any voltage, and any impedance (the AC equivalent of resistance) in an AC circuit[3] using complex numbers. If we do so, we find that nearly all the laws and properties learned for the analysis of DC circuits applies to AC circuits as well! In other words, *the reason we learn how to use complex numbers is so we can make AC circuit analysis as simple as DC circuit analysis.*

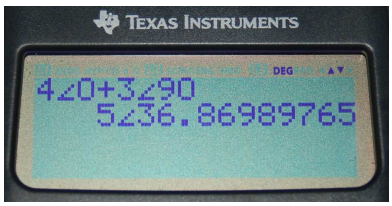
This use of complex numbers to represent and compute AC circuit quantities is called *phasor* mathematics. The name comes from the fact that the complex number represents not just the magnitude of the quantity but also its *phase relationship* to other quantities in the same circuit.

Polar notation is by far the most common complex-number notational form for representing real circuit quantities, because it directly relates to an oscilloscope's view of an AC voltage or current: the peak value of the whole wave in addition to the angle at some specific moment in time. Rectangular notation exists as an aid to addition and subtraction, because these two arithmetic operations happen to be easier to perform in rectangular form than polar form.

Thankfully, modern scientific calculators render these distinctions academic by providing convenient entry, display, and computation of complex numbers. This functionality makes complex-number arithmetic as easy as computing square roots, logarithms, trigonometric functions, or any other mathematical function commonly taken for granted. There are still some applications where you may need to switch from polar notation to rectangular in order to solve a particular problem, but this is a rare requirement (and the calculator will even perform this conversion for you!).

For example, consider these two series-connected AC voltage sources. The lower source outputs 4 Volts between points B and C, and serves as our "phase reference" with its angle arbitrarily set to 0^o. The upper source outputs 3 Volts between points A and B, and has a phase shift that is $+90^o$ ahead of (leading) the 4 Volt source. The "+" and "−" polarity symbols show instantaneous polarity when each source is at its 0^o point (i.e. the positive peak of its cosine wave):

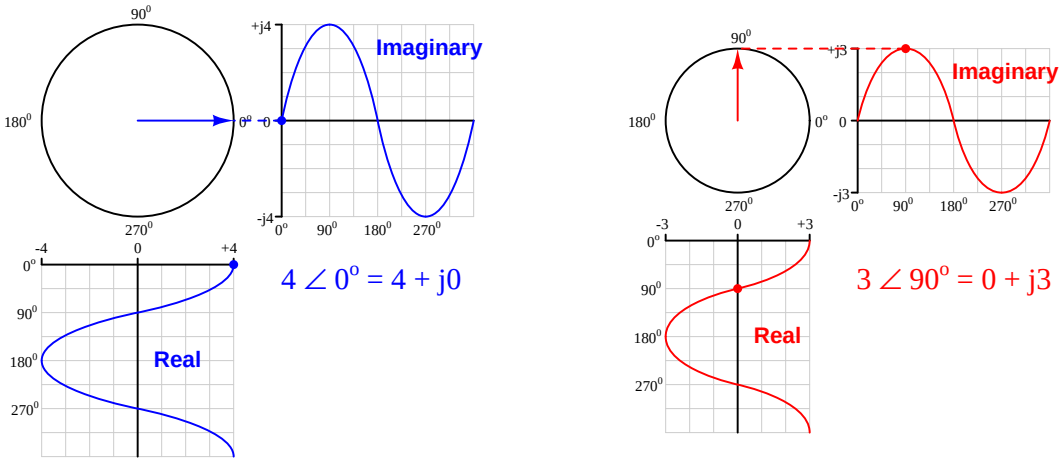




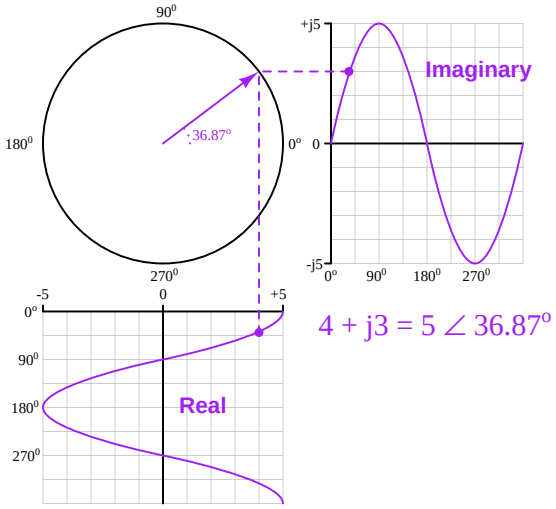
Kirchhoff's Voltage Law (KVL) applies to AC circuits just as well as to DC circuits, and therefore these two source voltages must *add* to create a total voltage between points A and C. The only difference between AC and DC voltages is that we must use complex numbers (phasors) to do the addition. The right-hand photograph shows the lower source's voltage value of 4 Volts $\angle\ 0^o$ added to the upper source's voltage value of 3 Volts $\angle\ 90^o$ using a Texas Instruments model *TI-36X Pro* hand calculator with its display mode set to complex-polar form. The calculator automatically performs all the trigonometric operations necessary to execute this complex-number addition.

[3] A few caveats apply, including the assumption that all voltages and currents share the same frequency.

If we only have access to a plain scientific calculator (i.e. lacking complex-number capability) we may still apply phasors to the calculation of this series network's total voltage. Complex-number quantities happen to be most easily added and subtracted in rectangular form, and best multiplied and divided in polar form. Since we know we need to add these two AC voltage phasors together, we will begin by expressing V_{AB} and V_{BC} as rectangular numbers. Mapping the polar values for V_{BC} (4 Volts $\angle$ 0^o) and V_{AB} (3 Volts $\angle$ 90^o) onto polar (circular) and rectangular (sine/cosine wave) graphs helps us visualize their conversions to rectangular form:



The sum of $4 + j0$ and $0 + j3$ is simply $4 + j3$. As it so happens, a phasor with polar magnitude of 5 and an angle of 36.87^o will have these equivalent real and imaginary values:



While it may be simple to plot horizontal and vertical phasors such as $4 \angle 0^o$ and $3 \angle 90^o$ onto graphs, it is far from obvious how to plot just the right phasor to give us a result having both non-zero real and imaginary values. Symbolic conversion from rectangular form into polar form requires some right-triangle trigonometry, as shown below:

$$\theta = \tan^{-1}\left(\frac{3}{4}\right) = 36.87^o$$

$$A = \sqrt{4^2 + 3^2} = \sqrt{16 + 9} = \sqrt{25} = 5 \text{ Volts}$$

This is the symbolic method for converting rectangular form into polar: use *arctangent* of the imaginary/real ratio to find the polar angle and use the *Pythagorean theorem* to find the polar magnitude. Therefore, an AC voltage of $4 + j3$ Volts is the same as an AC voltage of 5 Volts $\angle$ 36.87^o.

Incidentally, converting polar-form complex numbers into rectangular form relies on trigonometric functions as well. The real quantity of the rectangular number is the polar magnitude multiplied by the cosine of the angle, and the imaginary quantity is the polar magnitude multiplied by the sine of the angle. Applying these steps to our original V_{BC} and V_{AB} source voltages:

$$V_{BC} = 4\angle 0^o = 4\cos 0^o + j4\sin 0^o = 4 + j0$$

$$V_{AB} = 3\angle 90^o = 3\cos 90^o + j3\sin 90^o = 0 + j3$$

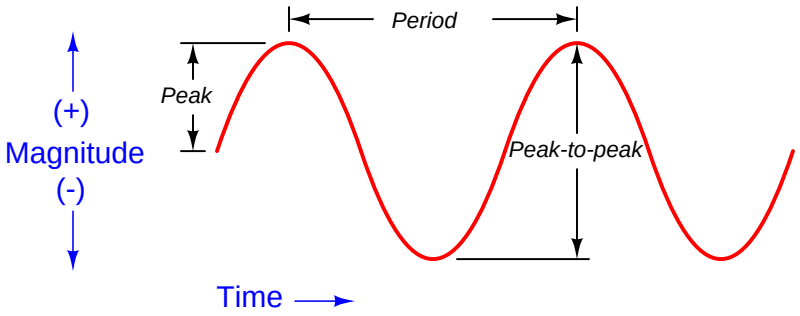
As you can see, these polar-rectangular-polar conversions are laborious compared to using a calculator that has complex-number capability.

Chapter 3

Full Tutorial

3.1 AC versus DC

Alternating current, or *AC*, is the designation for any electrical quantity varying periodically over time. Even though "current" is part of this phrase, it is often used to describe *voltage* as well. The tempo at which an AC quantity varies is called the *frequency*, mathematically symbolized by the variable f and measured in the unit of *cycles per second*[1] or *Hertz* (Hz). The magnitude of an AC quantity may be expressed in terms of the oscillation's *peak* value, its *peak-to-peak* value, or its equivalent DC value based on its ability to perform physical work (called the *Root-Mean-Square* or *RMS* value). The following illustration shows two complete "cycles" of a sine wave, with the vertical axis representing the magnitude of the AC quantity and the horizontal axis representing time:

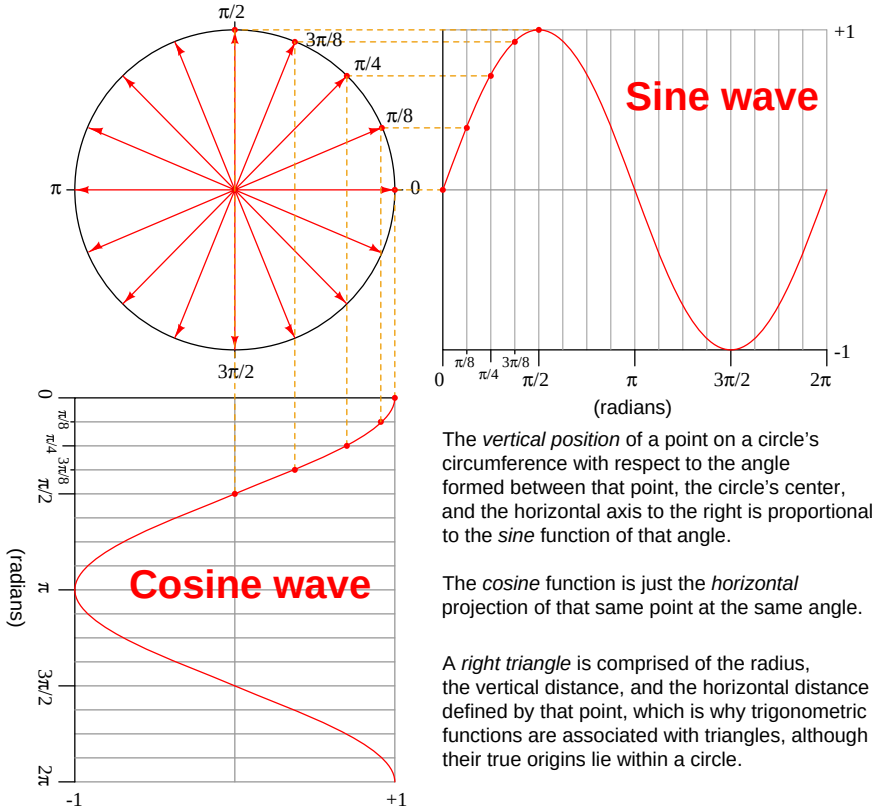


An AC voltage, for example, would possess a certain peak, peak-to-peak, and/or RMS voltage value as well as a period (measured in seconds) and a frequency (measured in cycles per second or Hertz). A greater magnitude appears "taller" on the vertical axis of a graph and a greater frequency appears "compressed" on the horizontal axis of a graph (i.e. more cycles within the same amount of time, or a shorter period).

A great many AC quantities are *sinusoidal* in shape; that is to say, their plot over time resembles that of a sine function or a cosine function over time. We must therefore begin our mathematical exploration of AC by reviewing some fundamental principles of trigonometry.

[1] Since a "cycle" is technically not a unit of measurement like the second, frequency is sometimes expressed in units of *inverse seconds*, or s^{-1}.

All trigonometric functions (e.g. sine, cosine, tangent) are based on the vertical and horizontal projections of a radius swept around a circle of constant diameter[2]. For this reason, one cycle of a sine-wave function represents one complete rotation around the circle.



The *vertical position* of a point on a circle's circumference with respect to the angle formed between that point, the circle's center, and the horizontal axis to the right is proportional to the *sine* function of that angle.

The *cosine* function is just the *horizontal* projection of that same point at the same angle.

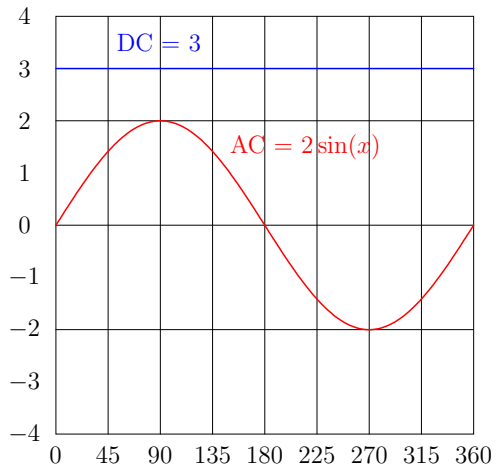
A *right triangle* is comprised of the radius, the vertical distance, and the horizontal distance defined by that point, which is why trigonometric functions are associated with triangles, although their true origins lie within a circle.

The angle of rotation is commonly expressed by mathematicians in *radians*: one radian being that angle associated with a section of the circle's circumference equal to the radius length. This is why 2π radians comprise one full rotation as shown in the illustration: because a full circle's circumference is 2π times longer than its radius. In engineering, the angular unit of the *degree* is more commonly used to express angles of rotation, there being 360 degrees in a full circle (and therefore 360 degrees for one complete cycle of a sine or cosine wave).

The frequency of a wave is simply an expression of rotational speed, since each "cycle" of the wave is one rotation around the defining circle. The unit of Hertz (cycles per second) therefore is equivalent to *revolutions per second*. Alternatively, frequency is sometimes referred to in terms of *radians per second*, with 1 Hertz being equivalent to 2π radians per second. When radians per second are the units of measurement, frequency is sometimes called *angular velocity* and represented by the lower-case Greek letter Omega (ω). Therefore, $\omega = 2\pi f$.

[2]Trigonometry is commonly associated with right triangles, which seem at first to have no relation to a circle. However, if one considers the radius of that circle drawn to a point on the circumference to be the hypotenuse of a right triangle inscribed within the circle, that triangle's opposite and adjacent side lengths represent the point's vertical and horizontal positions.

The analysis of DC circuits is possible by application of fundamental principles, such as the Conservation of Energy, the Conservation of Electric Charge, Ohm's Law, Joule's Law, and Kirchhoff's Voltage and Current Laws. AC circuits are no different in this respect because the fundamental elements of electric charge, voltage, and charge motion (current) remain the same. What *is* different, however, is the quantification of variables such as voltage and current existing in a continual state of change, as well as the arithmetic necessary to add, subtract, multiply, and divide these ever-changing quantities. Ohm's and Joule's Laws require multiplication and division; Kirchhoff's Laws require addition and subtraction. When quantities such as voltage, current, and resistance are constant these arithmetic operations are simple; when those same quantities continually vary these operations become considerably more complicated. Consider this graphical comparison of two quantities: a "DC" quantity with a constant value of 3, and an "AC" quantity being a sine wave with a peak value of 2:



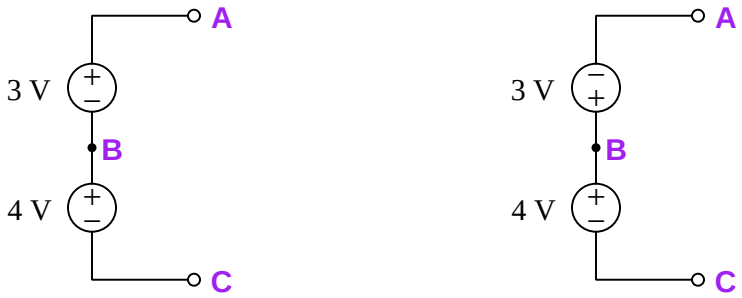
Since the majority of AC circuit variables are sinusoidal[3] in nature, we will focus on the mathematical treatment of sine waves, beginning with addition and subtraction. Although AC circuits with non-sinusoidal voltages and currents exist, it is mathematically possible[4] to express *any* repetitive oscillation of *any* shape as a summation of multiple sine and cosine waves, which means any mathematical techniques capable of handling sinusoidal functions may be extended to handle non-sinusoidal functions as well.

[3] The AC electricity created by an electromechanical generator oscillates in a sinusoidal pattern due to the circular motion of the generator's magnetized rotor. Thus, the AC voltage produced by a spinning generator is an expression of the same phenomenon of sine and cosine waves being projections of a spinning radius. In fact, if an AC generator is equipped with two sets of stationary windings exposed to the spinning rotor's magnetic field, each winding set 90 degrees apart from the other, one winding will output a sine wave while the other outputs a cosine wave.

[4] The mathematical technique for expressing a non-sinusoidal function as a series of sinusoidal functions is called the *Fourier Transform*.

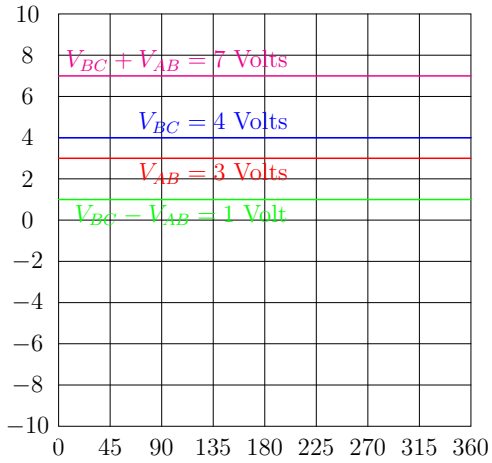
3.2 Additive quantities

Let us consider a pair of DC voltage sources connected in series with each other, reviewing how we should compute the "total" voltage of the source pair. In one scenario the two sources are aiding each other, and in the other scenario they oppose each other:



In the left-hand network where the two sources aid each other, the total voltage measured at point A with reference to point C (e.g. a voltmeter's red test lead on A and black test lead on C), or V_{AC}[5], should be +7 Volts. In the right-hand network where the two sources oppose each other, the total voltage measured at point A with reference to point C (e.g. a voltmeter's red test lead on A and black test lead on C), or V_{AC}, should be +1 Volt. This is all in accordance with Kirchhoff's Voltage Law.

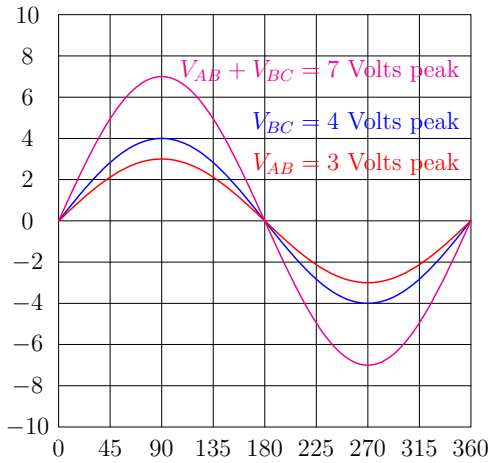
We may represent both scenarios graphically, showing how the two DC voltage quantities either add or subtract to make the resultant voltage V_{AC}:



The sum of the 4 Volt line and the 3 Volt line is a 7 Volt line; the difference between the 4 Volt line and the 3 Volt line is a 1 Volt line.

[5]It is worth noting that "AC" is used here to identify test points, and has nothing to do with *AC* as used to abbreviate the phrase *Alternating Current*.

Let us consider the same series-connected voltages, but this time with those sources both being AC rather than DC. For the sake of illustration, we will consider source V_{BC} to have a peak value of 4 Volts, and source V_{AB} to have a peak value of 3 Volts, both at the same frequency. These sinusoidal plots were created using a computer program written in the `C++` programming language, the computer evaluating the formulae $3 \sin x$ and $4 \sin x$ for values of x ranging from 0 degrees to 360 degrees. The sum was similarly calculated via computer[6], adding the two sine waves' values together for every calculated value of x (sum $= 3 \sin x + 4 \sin x$):



Here we clearly see a sine wave with a peak value of 3 (V_{AB}) along with a sine wave with a peak value of 4 (V_{BC}), added together to make a third sine wave having a peak value of 7. This is rather intuitive: if we imagine these two sine waves adding together at every point in time, the sum must be zero at that exact angle when both waves cross zero, and the sum should peak at 7 Volts when each source is at its respective peak. Algebraic factoring proves this as well:

$$V_{AB} = 3 \sin x \qquad V_{BC} = 4 \sin x$$

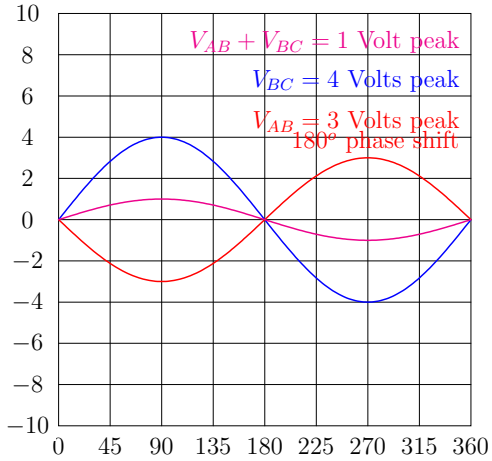
$$V_{AB} + V_{BC} = 3 \sin x + 4 \sin x$$

$$V_{AB} + V_{BC} = (3 + 4) \sin x$$

$$V_{AB} + V_{BC} = 7 \sin x$$

[6] There is no reason why this sum could not have been calculated by hand, using trigonometric tables or an electronic calculator to sum the respective values of two sine functions over a domain of 0 to 360 degrees. Computers merely expedite this laborious task.

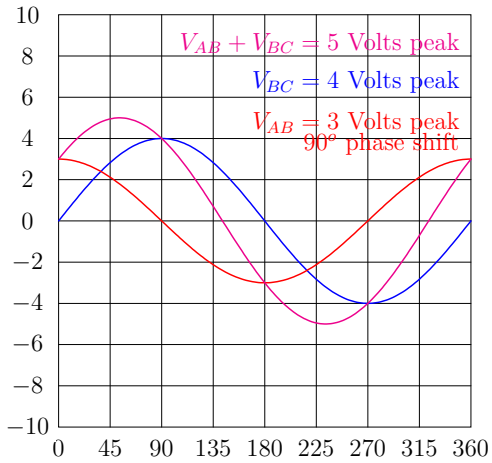
If we were to program the computer to plot the sum of two AC voltages *opposing* each other (i.e. the two waveforms 180 degrees out of phase with each other), the result is not surprising either:



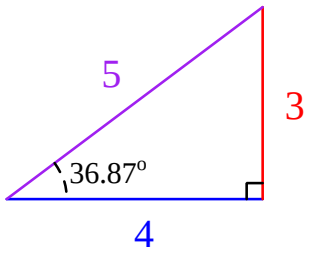
Again, the computer is simply plotting two sine waves, $4 \sin x$ and $3 \sin(x + 180^o)$, and adding their respective values together at every computed value of x from 0 to 360 degrees to plot the third wave. The result is another sine wave with a peak value of 1. When x is equal to 0 degrees, 180 degrees, or 360 degrees, each of the two sine waves' values is zero and therefore the summation of those waves is also zero at those points. When x is equal to 90 degrees or 270 degrees, each of the sine waves' values reaches its peak (with opposite signs), resulting in a summation equal to -1 or $+1$.

It should be clear from these two graphical examples that the sum of two in-phase sine waves is analogous to the sum of two series-aiding DC voltages, while the sum of two sine waves precisely 180 degrees out of phase with each other is analogous to the sum of two series-opposing DC voltages. This much is simple, and appears no more complicated than any DC calculation. What is not clear yet, though, is what happens when we set the phase shift at some value other than 0 degrees or 180 degrees, because this condition has no DC analogue. In DC, voltage sources may (only) aid or oppose each other. To consider an AC example with a phase shift other than 0 degrees or 180 degrees is to consider a case where two voltage sources neither directly aid nor directly oppose each other.

Let us explore the summation of two sine waves phase-shifted from each other by 90 degrees. We will use the same peak amplitudes as before, 4 Volts peak and 3 Volts peak, programming the computer to add each of those waves' values together at every computed value of x:



The result, according to the graph, is a sine wave with a peak value of *5 Volts* and a phase shift that is approximately 40 degrees ahead of the 4 Volt (blue) wave. This result is quite different from the 7 Volt total we got when the voltage waves were in-phase with each other, and also different from the 1 Volt total we got when the two AC voltages were perfectly out of phase with other. However, those readers familiar with trigonometry should recognize the values 3, 4, and 5 as one of the unique sets of integer values that work as side-lengths for a right triangle:



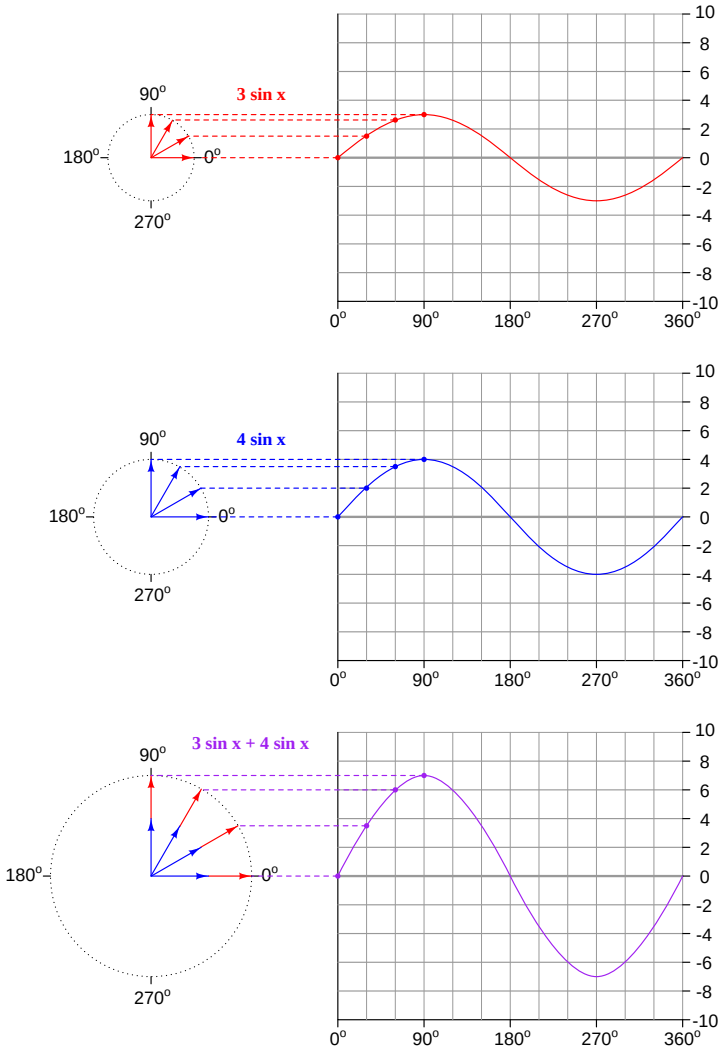
This is how two phase-shifted quantities add together: represented as line-segments having length (magnitude) and direction (angle), the resultant (sum) is another line-segment joining the starting-point to the end-point.

An important lesson to be learned here is that when adding together two sine waves that are either perfectly in-step with each other or perfectly inverted from each other, we may calculate the resultant sum simply by using the scalar[7] peak values of those waves. However, when we try adding waves that are out-of-step with each other by some angle other than 180 degrees, simple scalar measurements of peak values lack all the information necessary to compute that sum. A 4 Volt peak sine wave added to a 3 Volt peak sine wave creates a 5 Volt peak sine wave *only* when the 4-Volt and 3-Volt waves are shifted 90 degrees apart in phase.

[7] A *scalar* quantity is one having a single dimension.

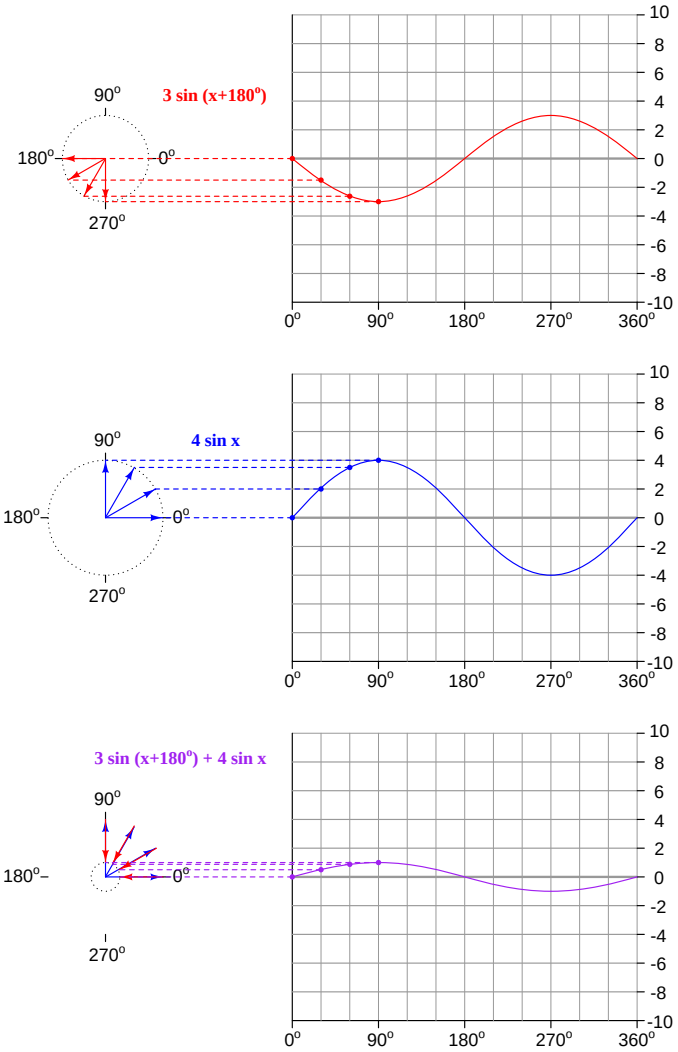
This similarity between a 3-4-5 right triangle and the sum of sine waves (peak of 3 plus peak of 4 equaling peak of 5) is no coincidence. Recall that sine waves are really just the projection of a point's vertical position as it sweeps around the circumference of a circle. The sum of any two sine waves at the same frequency, therefore, must be the sum of those height-projections at every respective point around the circles.

The following illustrations show the two AC voltage sources' sine waves as projections of circular radii. Each of the radii are called *phasors*, having respective lengths of 3 and 4. At the bottom is a summation of these phasors, forming a circle of radius 7 and a corresponding sine wave with a peak value of 7:



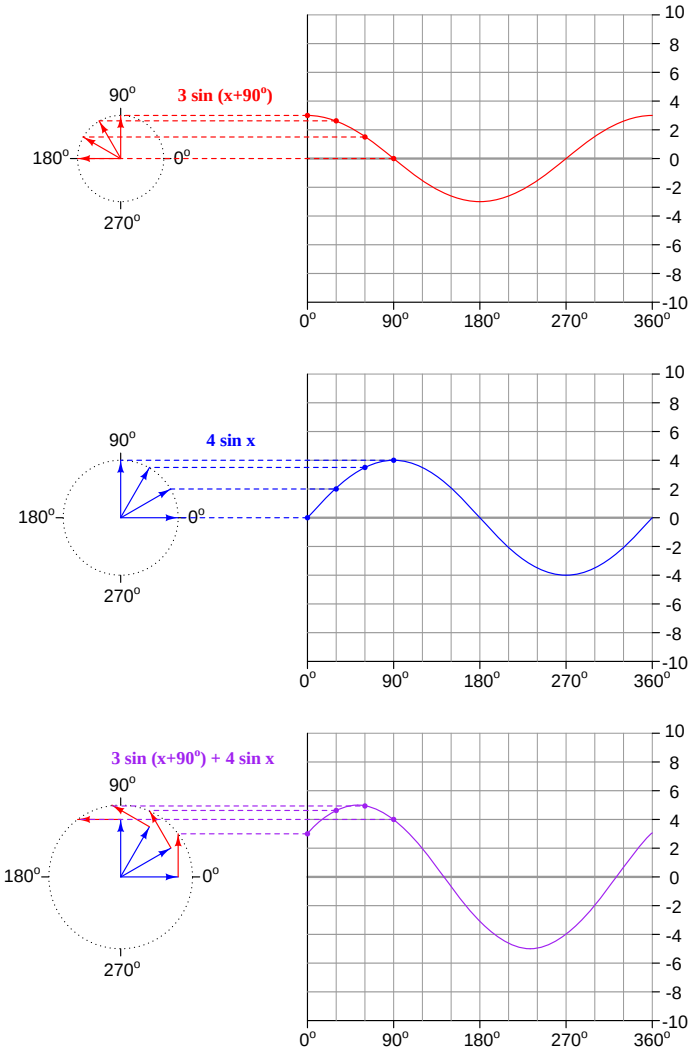
This graphically represents our first AC scenario, where a 3 Volt sine wave and a 4 Volt sine wave at the same frequency and in-phase with each other add together to make a 7 Volt sine wave.

Now let us analyze our second scenario by the same graphical method, projecting the sum of a 4 Volt sine wave and a 3 Volt sine wave that is 180 degrees out of phase:



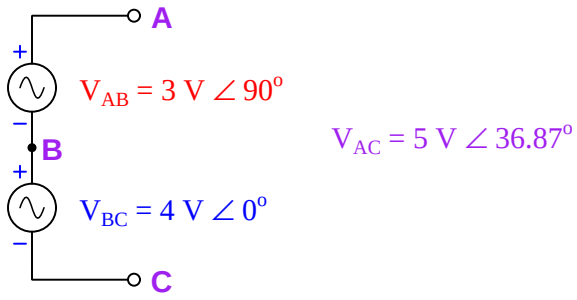
Note how the phasors of the 3 Volt sine wave begin at 180 degrees and end at 270 degrees because that function is offset 180 degrees from the 4 Volt sine wave. The sum of these two sine waves, of course, is a sine wave having a peak value of only 1, since for every value of x the two phasors are pointing in opposite directions of each other and therefore subtract.

Finally, we will analyze our third scenario by showing the graphical sum of a 4 Volt sine wave and a 3 Volt sine wave that is 90 degrees out of phase:

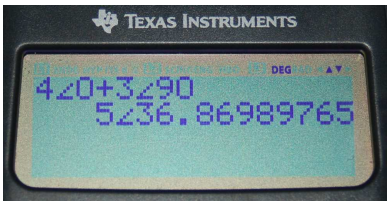


Pay close attention to lower illustration, where the circle's radius is defined by the position of the point at the end of two stacked phasors, one having a length of 4 and the other with a length of 3, offset from each other by 90 degrees. This circle has an effective radius of 5, because its radius is the hypotenuse of a 3-4-5 right triangle inscribed within the circle. Note also how the projected function exhibits its own unique phase shift, starting neither at zero like the $4 \sin x$ wave nor starting at the positive peak like the $3 \sin(x + 90^o)$ wave. The phase offset of this summation function is equal to the angle between the 4 and 5 sides of a 3-4-5 triangle (36.87 degrees), which means we may write it symbolically as $5 \sin(x + 36.87^o)$. Therefore, $3 \sin(x + 90^o) + 4 \sin x = 5 \sin(x + 36.87^o)$.

Now that we have seen the graphical relationships between added sinusoid functions and how these apply to AC circuit quantities that add together (e.g. series-connected AC voltage sources), we must find some way to perform these calculations for the purpose of AC circuit analysis. Fortunately for us, many electronic calculators provide such functionality in the form of *complex number arithmetic*. A sinusoidal function having a peak value and a phase angle may be expressed in *polar form* as a complex number's[8] *magnitude* and *angle*. Consider the following two AC voltage sources connected in series, one at 4 Volts peak and the other at 3 Volts peak (with a 90 degree shift), the "+" and "−" polarity marks showing peak voltage at each source's 0^o moment in time. We may compute total series voltage by simply adding the polar-form complex numbers $4\angle 0^o$ and $3\angle 90^o$.



The following photograph shows this calculation performed on a Texas Instruments model *TI-36X Pro* hand calculator, with its display mode set to complex-polar form:



Calculations based on the treatment of electrical quantities as complex numbers are not limited to addition and subtraction. Complex numbers may be multiplied, divided, squared, square-rooted, etc. just like normal "real" numbers, which means we are able to perform calculations with AC circuit quantities just the same as with DC circuit quantities, using all the same laws and principles[9].

The fact that all these complex number operations are performed as easily as normal "real" numbers using a suitable hand calculator means we have a powerful tool for AC circuit analysis. If performing AC circuit calculations using Ohm's Law and Kirchhoff's Laws is your only goal, then there is nothing more you need to learn about this topic than where to obtain a suitable hand calculator capable of performing complex number arithmetic in polar form. The rest of this tutorial is dedicated to a deeper exploration of complex numbers and their relation to sinusoidal functions.

[8] A "complex" number is a combination of a "real" quantity and an "imaginary" quantity. Complex numbers may be expressed in *rectangular* form where the real and imaginary portions are separately shown, or they may be expressed in *polar* form where the phasor's length and angle are shown.

[9] A notable exception is the calculation of power by multiplying voltage and current, which doesn't work quite the same for AC as it does for DC. The calculation of power in AC circuits is a topic unto itself, covered in a different module.

3.3 Complex numbers and exponential functions

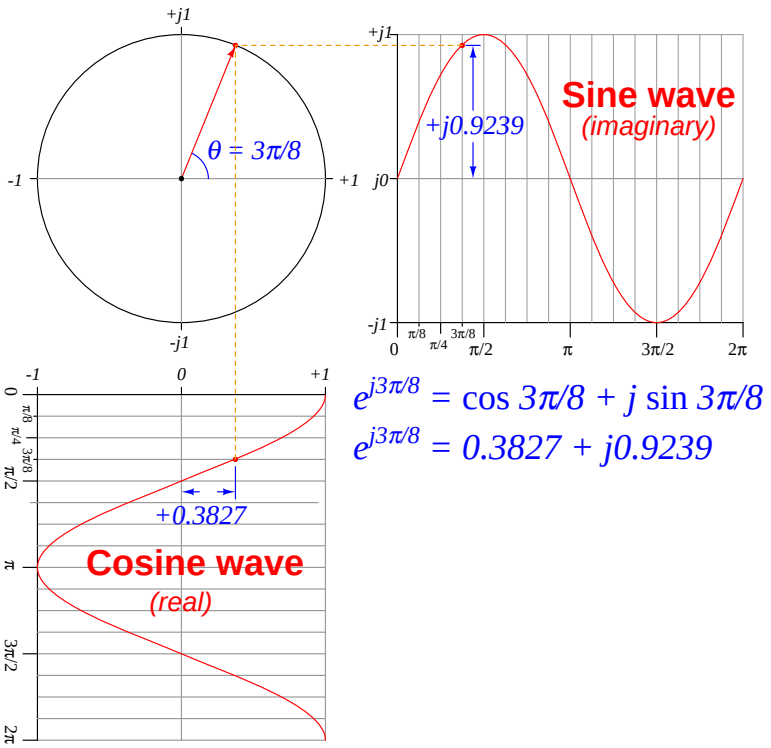
Complex numbers and sinusoidal functions are related by a famous mathematical formula called *Euler's Relation*, discovered by the Swiss mathematician Leonhard Euler (1707-1783):

$$e^{j\theta} = \cos\theta + j\sin\theta$$

Where,

e = Euler's number (approximately equal to 2.718281828)
θ = Angle of phasor, in radians
$\cos\theta$ = Horizontal projection of a unit phasor (along a real number line) at angle θ
j = Imaginary "operator" equal to $\sqrt{-1}$, alternatively represented as i
$j\sin\theta$ = Vertical projection of a unit phasor (along an imaginary number line) at angle θ

To illustrate, we will apply Euler's relation to a unit[10] phasor having an angular displacement of $\frac{3\pi}{8}$ radians (67.5 degrees):



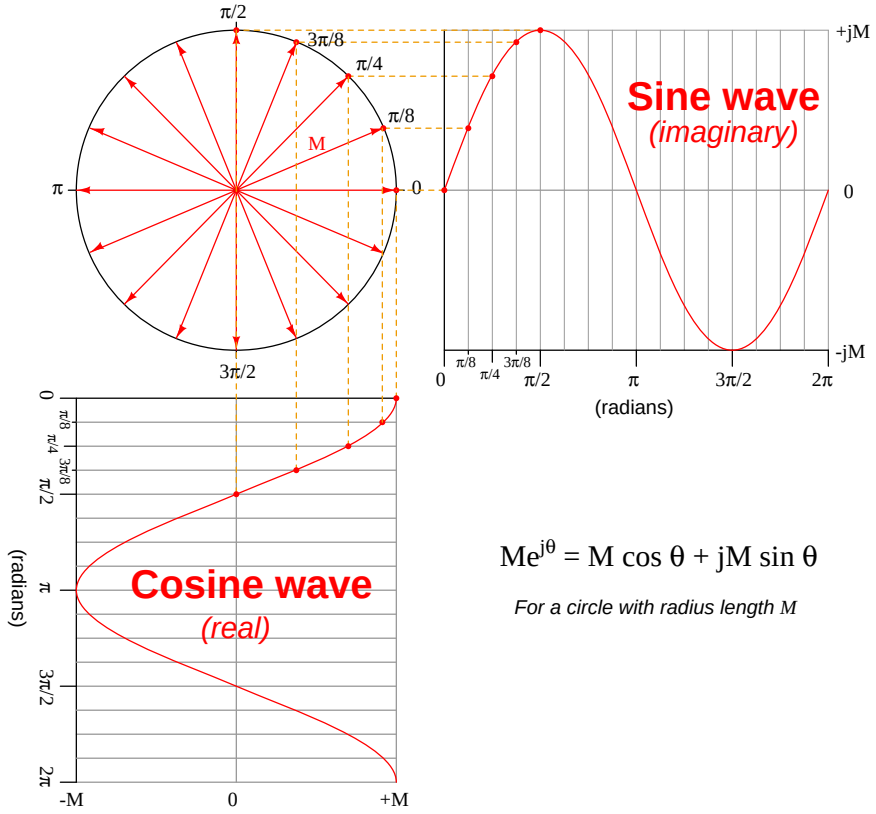
The sine wave exists as the vertical projection of the rotating phasor on the "imaginary" (j or i) axis, while the cosine wave exists as the horizontal projection of that same phasor on the "real" axis. At the particular angle arbitrarily chosen for this example ($3\pi/8$ radians) the vertical quantity happens to be $j0.9239$ while the horizontal quantity happens to be 0.3827.

[10] The term "unit phasor" simply refers to a phasor with a length of 1 ("unity").

For "unit" circles having a radius value of 1, Euler's Relation perfectly describes the projected cosine and sine functions for any angle θ. If our goal, though, is to represent voltage and current quantities with peak values other than 1, we must append a multiplication factor to Euler's Relation accounting for the *magnitude*[11] (M) of the sinusoidal function:

$$Me^{j\theta} = M\cos\theta + jM\sin\theta$$

Using this version of Euler's Relation, we may perfectly describe the functions of a circle having a radius length of M:



[11] The terms *amplitude* and *magnitude* are often used synonymously, but here we are using "amplitude" to describe the intensity of the circuit quantity (e.g. voltage or current) at any given moment in time and "magnitude" to describe only the peak value it reaches.

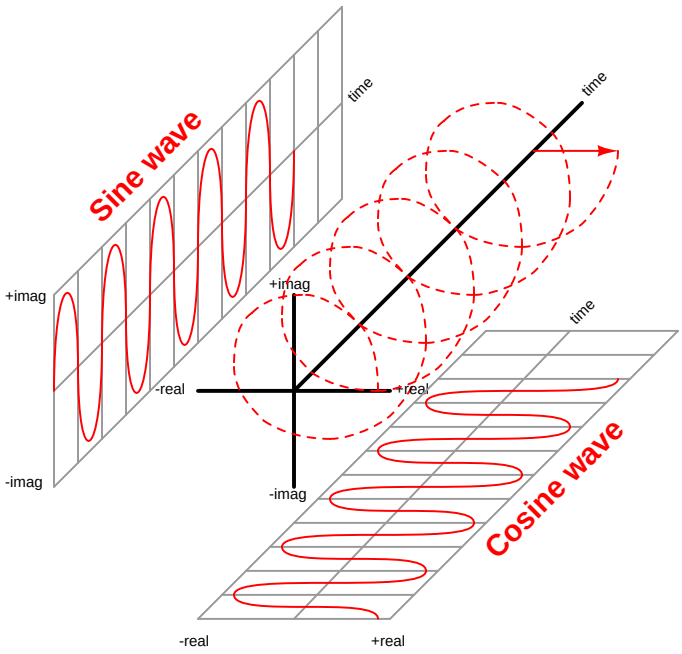
An important extension of Euler's Relation to real-world AC circuit applications is the inclusion of *time* as one of the variables, since AC quantities oscillate over time. Recall that the *angular velocity* of a rotating phasor (ω) is equivalent to its frequency measured in radians per second rather than Hertz. The angle of a phasor at any point in time, therefore, is the product of its angular velocity (radians/second) and the specific value of time (seconds):

$$\theta = \omega t \qquad \text{[radians]} = \text{[radians/second][seconds]}$$

Therefore, we may substitute ωt in place of θ in Euler's Relation to now describe phasors and their sinusoidal projections as functions of time:

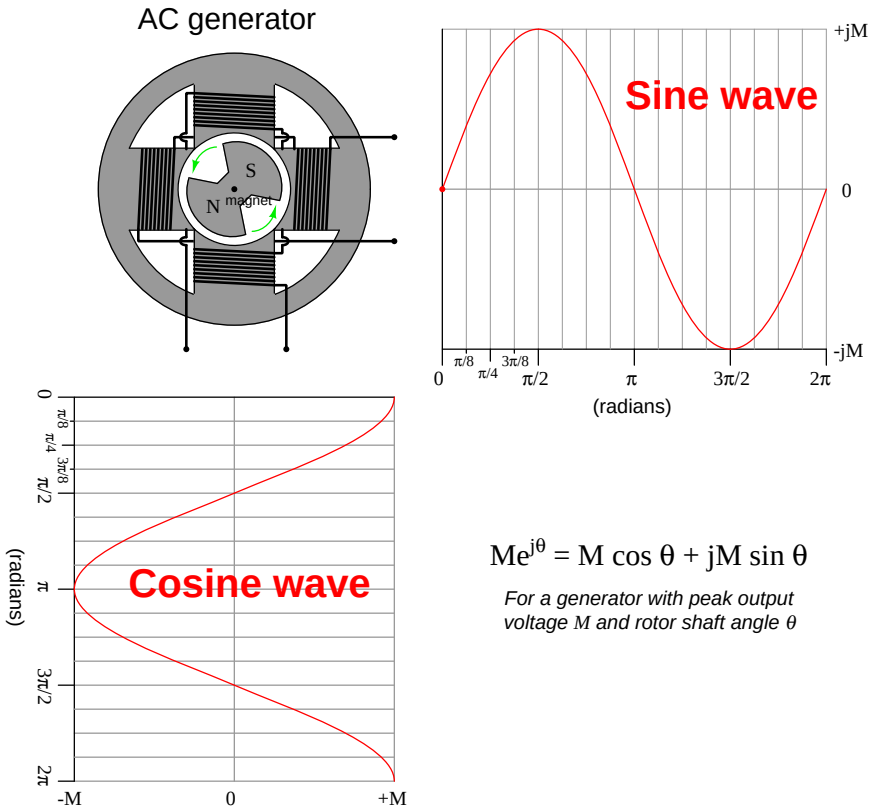
$$Me^{j\omega t} = M\cos(\omega t) + jM\sin(\omega t)$$

This too has a graphical representation, with the real, imaginary, and time axes all perpendicular to each other, forming a three-dimensional graph:



The phasor's tip traces a spiral path as it rotates around the circle and progresses along the time axis. When viewed from the end (with the time axis pointing away from us), all we see is the phasor's rotation. When viewed from the side, we see either a cosine wave or a sine wave depending on whether we are looking at the phasor's horizontal projection or its vertical projection. A physical model of this is either a compression-style coil spring or a corkscrew: viewed from the end you see a circle; viewed from the side you see a sinusoid.

This three-dimensional graph has an electrical analogue as well: an AC generator with a spinning magnetic rotor (representing the rotating phasor) and two sets of stationary coils ("windings"), one winding generating a sine wave and the other generating a cosine wave:



The angular offset between these two sets of stator windings accounts for the 90 degree ($\frac{\pi}{2}$ radian) phase shift between the two sinusoidal output voltages. One winding pair experiences the spinning rotor's magnetic field one-quarter rotation before the other winding pair. The two sets of windings essentially "view" the spinning rotor from different perspectives. Euler's Relation predicts the voltage developed at each winding based on the maximum (peak) voltage generated by either winding (M) and the angular position of the rotor at that instant in time (θ).

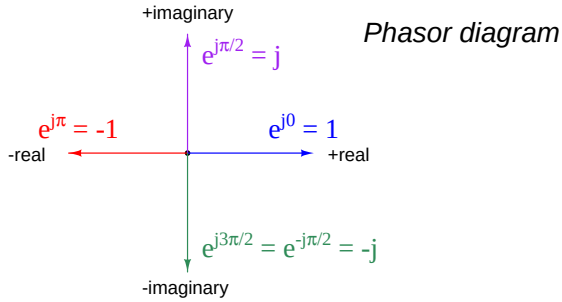
A surprising amount of electrical engineering is founded on Euler's Relation. The equivalence between sinusoidal functions ($\cos\theta + j\sin\theta$) and exponential functions ($e^{j\theta}$) greatly simplifies a number of different mathematical analyses, and forms the basis for the rules we will use to add, subtract, multiply, and divide AC circuit quantities.

Although Euler's Relation is the basis of AC circuit mathematics due to its ability to completely represent a phasor at any point in time, the full form of Euler's Relation is often omitted for simplicity's sake. In its place, electrical practitioners often use simplified mathematical notation to describe phasors. The two sides of Euler's Relation – the *exponential* ($Me^{j\theta}$) side and the *trigonometric* ($M\cos\theta + jM\sin\theta$) side – are instead referred to as *polar* and *rectangular*, respectively. "Polar" refers to the description of a phasor on a polar plot: an arrow pointing away from the plot's center (pole) with a specified length M and angle from horizontal θ. "Rectangular" refers to the description of a phasor's tip on a rectangular plot: the "real" term $M\cos\theta$ describing its position on the horizontal axis and the "imaginary" term $jM\sin\theta$ describing its position on the vertical axis. Shorthand phasor notation consists of the phasor's magnitude followed by an angle symbol ($\angle$) and the angle value for polar notation, and the real and imaginary coordinates for rectangular notation:

Full notation	$Me^{j\theta}$	$M\cos\theta + jM\sin\theta$
Shorthand notation	$M\angle\theta$	$x + jy$

For illustrative purposes we will consider all these phasor representations at an magnitude (M) of one and at four different angle values, one for each axis: 0 radians, $\frac{\pi}{2}$ radians, π radians, and $\frac{3\pi}{2}$ radians (corresponding to 0, 90, 180, and 270 degrees, respectively). All four examples are shown in written form as well as graphical (i.e. a *phasor diagram*):

Angle (θ)	**Exponential**	**Trigonometric**	**Rectangular**	**Polar**
0 radians $= 0^o$	e^{j0}	$\cos 0 + j\sin 0$	$1 + j0 = 1$	$1 \angle 0^o$
$\pi/2$ radians $= 90^o$	$e^{j\pi/2}$	$\cos 90^o + j\sin 90^o$	$0 + j1 = j$	$1 \angle 90^o$
π radians $= 180^o$	$e^{j\pi}$	$\cos 180^o + j\sin 180^o$	$-1 + j0 = -1$	$1 \angle 180^o$
$3\pi/2$ radians $= 270^o$	$e^{j3\pi/2}$	$\cos 270^o + j\sin 270^o$	$0 - j1 = -j$	$1 \angle 270^o$



The third example – at an angle of π radians (180 degrees) – is uniquely elegant because it relates several important mathematical constants (e, i, π, 1, and 0) in a single formula. Here we will use the common mathematical symbol for an imaginary number (i) instead of the symbol typically used by electrical practitioners (j):

$$e^{i\pi} = -1 \qquad \text{or} \qquad e^{i\pi} + 1 = 0$$

The four basic arithmetic operations (addition, subtraction, multiplication, and division) are shown here in general form using two phasors, one with magnitude M and angle x, and the other with magnitude N and angle y:

$$Me^{jx} + Ne^{jy} = (M\cos x + N\cos y) + j(M\sin x + N\sin y)$$

$$Me^{jx} - Ne^{jy} = (M\cos x - N\cos y) + j(M\sin x - N\sin y)$$

$$Me^{jx} \times Ne^{jy} = MNe^{j(x+y)}$$

$$Me^{jx} \div Ne^{jy} = \frac{M}{N}\left[e^{j(x-y)}\right]$$

To give a more concrete example, we may perform all these arithmetic operations on the phasors $4\angle 0^o$ and $3\angle 90^o$, using shorthand notation:

$$4\angle 0^o + 3\angle 90^o = 4\cos 0^o + 3\cos 90^o + j(4\sin 0^o + 3\sin 90^o) = 4 + j3$$

$$4\angle 0^o - 3\angle 90^o = 4\cos 0^o - 3\cos 90^o + j(4\sin 0^o - 3\sin 90^o) = 4 - j3$$

$$4\angle 0^o \times 3\angle 90^o = 12\angle 90^o$$

$$4\angle 0^o \div 3\angle 90^o = \frac{4}{3}\angle -90^o$$

Addition and subtraction in polar form necessitates the use of sine and cosine functions. If, however, the phasors in question are already cast in rectangular form, addition and subtraction is seen to be nothing more than combining the real and imaginary terms. Repeating the first two lines with $4\angle 0^o$ expressed as $4 + j0$ and $3\angle 90^o$ expressed as $0 + j3$:

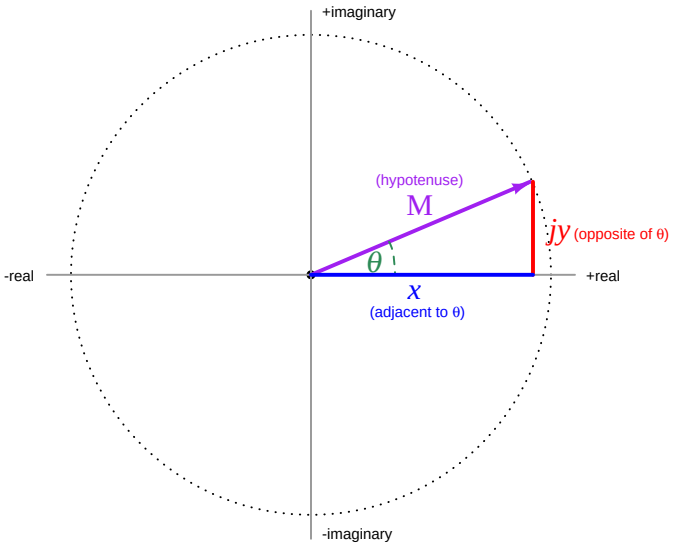
$$(4 + j0) + (0 + j3) = (4 + 0) + (j0 + j3) = 4 + j3$$

$$(4 + j0) - (0 + j3) = (4 - 0) + (j0 - j3) = 4 - j3$$

It should be clear from these examples that addition and subtraction are best performed in rectangular form while multiplication and division lend themselves best to polar form.

3.4 Trigonometric conversions

If we ever find ourselves needing to convert from rectangular form into polar form, we may do so using standard trigonometric methods. All we need to do is regard the real and imaginary portions of the rectangular-form phasor as side-lengths of a right triangle inscribed within a circle of radius M equal to the magnitude (i.e. the greatest amplitude value reached by the wave at any point) of the phasor in question:



The polar angle θ may be calculated three different ways, based on side y of the triangle being *opposite of*[12] angle θ and side x being *adjacent to* angle θ, knowing that the tangent function is the ratio between opposite and adjacent, the cosine function is the ratio between adjacent and hypotenuse, and that the sine function is the ratio between opposite and hypotenuse:

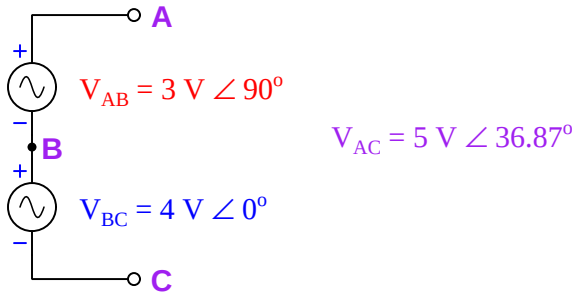
$$\theta = \tan^{-1}\left(\frac{y}{x}\right) \qquad \theta = \cos^{-1}\left(\frac{x}{M}\right) \qquad \theta = \sin^{-1}\left(\frac{y}{M}\right)$$

The polar magnitude M may be directly calculated from side lengths x and y using the Pythagorean Theorem:

$$M = \sqrt{x^2 + y^2}$$

[12] A misconception I've often encountered with students of trigonometry is that they assume the vertical side of a right triangle must always be the "opposite" and the horizontal side must always be the "adjacent". This is not necessarily true, and in fact is only true when the angle in question is the one between the horizontal side and the hypotenuse. The terms "opposite" and "adjacent" refer to the particular angle being referenced. If we were to consider the angle between the M and y sides of our triangle, for example, then x would be the side opposite that angle and y would be the side adjacent to that angle.

Now that we have explored the arithmetic of phasors, we may return to our problem of calculating the total voltage from two series-connected AC voltage sources. The result has already been determined by graphical solution (a computer summing all respective points of $4 \sin x$ and $3 \sin(x + 90^o)$ to yield a sine wave having an magnitude of 5) as well as by hand calculator (the TI-36X Pro adding $4\angle 0$ and $3\angle 90$ to arrive at $5\angle 36.86989765$), and so we show it again in the illustration:



Let us calculate this same sum without the benefit of a computer plotting hundreds of points on a graph or a hand calculator capable of performing complex-number arithmetic. Since we know the electrical principle at work here is Kirchhoff's Voltage Law, we know the total voltage V_{AC} must be equal to the sum of the voltages V_{BC} and V_{AB}. We also know that phasor sums are most easily computed in rectangular form, and so our first task is to convert the two voltage source values from polar form into rectangular form using cosine and sine functions:

$$V_{BC} = 4\angle 0^o = 4 \cos 0^o + j4 \sin 0^o = 4 + j0$$

$$V_{AB} = 3\angle 90^o = 3 \cos 90^o + j3 \sin 90^o = 0 + j3$$

The total voltage of these two series-connected sources must therefore be:

$$V_{BC} + V_{AB} = (4 + j0) + (0 + j3) = 4 + j3$$

The result, $4+j3$, while mathematically correct, does not relate directly to the indication given by a voltmeter connected between points A and C. Polar form would be a more realistic representation, giving us the magnitude (peak amplitude) value of the AC voltage as well as the phase shift angle. Therefore, our final step is to convert rectangular form into polar form:

$$\theta = \tan^{-1}\left(\frac{3}{4}\right) = 36.87^o$$

$$\text{Voltage magnitude} = \sqrt{4^2 + 3^2} = \sqrt{16 + 9} = \sqrt{25} = 5 \text{ Volts}$$

An AC voltage having an magnitude of 5 Volts and a phase angle of 36.87^o is our result, and therefore $V_{AC} = 5\angle 36.87^o$.

3.5 Summary

The benefit of phasor mathematics is that it gives us a tool to apply fundamental laws and principles of DC circuits to AC circuits. This is incredibly important, as without such a tool we would have to approach AC circuits quite differently than DC circuits. As it is, all we need is a new way to perform the basic arithmetic operations of addition, subtraction, multiplication, and division incorporating magnitude and phase shift, and we are able to re-purpose all the concepts learned on DC circuits to AC circuits. It is the goal of scientific exploration to *unify* knowledge by seeking commonalities between seemingly disparate phenomena. Phasor mathematics is such a bridge, unifying DC and AC circuit analysis.

All mathematical techniques have practical limitations, though, and so we must outline some caveats of phasor arithmetic:

- The phasor arithmetic examples we've explored all assume a constant and equal frequency for all quantities. A 4 Volt source at 0 degrees adds to 3 Volt source at 90 degrees to make 5 Volts at 36.87 degrees, *but only if those two voltage sources output precisely the same frequency.* If the two quantities in question have differing frequencies, it means their phasors rotate around the circle at different speeds, and therefore their relative phase angles will not be constant.

- The phase angle specified for any AC waveform (i.e. voltage or current) is always *relative* to some arbitrary reference, and not absolute. For the 4 Volt and 3 Volt sources we keep using as an example, what we mean when we say the 4 Volt source has an angle of 0 degrees and the 3 Volt source has an angle of 90 degrees is that the 4 Volt source is our *phase reference* for the circuit. The 3 Volt source's 90 degree angle simply means that source is 90 degrees ahead of the reference source at all times. We could have made the 3 Volt source our phase reference at 0 degrees and specified the 4 Volt source as having a phase angle of -90 degrees, and the sum would still be 5 Volts.

- Multiplication or division of two sinusoids yields a result with a different frequency. This is what makes power calculations complicated in AC circuits. Joule's Law ($P = IV$) requires multiplying current by voltage, which means the frequency of power in an AC circuit is actually different than the frequency of the voltage or current! The magnitude of the calculated power will be accurate, but the phase angle of that product of voltage and current will not refer to a constant phase shift as it does between the voltage and current waveforms, and so will not have the same meaning. We will explore this topic in greater detail in another module.

- In developing our mathematical definition of a phasor – showing the rotating radius project as a sinusoidal waveform – the phasor's length represents the magnitude of that quantity. For example, a voltage specified as $3\angle 90^o$ according to this definition has a magnitude of 3 Volts *peak*. In practice this definition is not strictly adhered to. In fact, it is quite common for electrical practitioners to specify the magnitude of all phasors in a circuit as Volts and Amperes *RMS* rather than peak. So long as every single phasor in that circuit is specified the same way, this is not a problem. Confusion results when some phasors are expressed in peak Volts or Amperes while others are assumed to express RMS Volts or Amperes.

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 Derivation of Euler's Relation

Leonhard Euler was a Swiss mathematician who lived from 1707 to 1783, and was responsible for discovering what we now call *Euler's Relation*, a mathematical formula relating exponential functions (e^x) to trigonometric functions ($\sin x$, $\cos x$). This relation forms the basis for phasor arithmetic, which is an incredibly useful tool for simplifying calculations in AC circuits where voltages and currents take the form of sinusoidal functions.

To begin, we must introduce the concept of a mathematical *series*, which is a sum of successive terms. The symbol used to denote a sum of many terms is the Greek capital letter "sigma" (Σ), with labels below and above specifying the beginning and end of the series. The following example is a form of "geometric" series of x to the n power, with n ranging from zero to infinity:

$$\sum_{n=0}^{\infty} x^n$$

Expanded, it is equivalent to the following expression:

$$\sum_{n=0}^{\infty} x^n = x^0 + x^1 + x^2 + x^3 + x^4 + \cdots + x^\infty$$

Infinite series may be approximated using digital computers to perform the term calculations and summation, with the result approaching ever closer to the true value with every iteration.

Another important mathematical concept necessary to understand Euler's Relation is that of the *factorial*. Symbolized by an exclamation mark, a "factorial" is the product (multiplication) of all whole numbers from the starting value down to one. Shown here are some factorial examples to illustrate:

- $0! = 1$ (by definition)
- $1! = 1 \times 1 = 1$
- $2! = 2 \times 1 = 2$
- $3! = 3 \times 2 \times 1 = 6$
- $4! = 4 \times 3 \times 2 \times 1 = 24$
- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$

Euler's Constant, symbolized by the letter e, happens to be the result of an infinite series of factorials:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

The partial sum (up to $n = 7$) appears as follows:

$$\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!}$$

$$1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \frac{1}{720} + \frac{1}{5040}$$

It should be clear to see how each successive term becomes dramatically smaller than the one before it, which means the sum will asymptotically approach some final value. In this case, the final value happens to be e. If we examine a table of figures showing the summation of these terms from $n = 0$ to $n = 7$, the convergence becomes clear:

n	$n!$	$\frac{1}{n!}$	$\Sigma\frac{1}{n!}$
0	1	1	1
1	1	1	2
2	2	0.5	2.5
3	6	0.16667	2.66667
4	24	0.04167	2.70833
5	120	0.00833	2.71667
6	720	0.00139	2.71806
7	5040	0.00020	2.71825

The value of e is approximately 2.718281828459045, as we can see here how closely the series approaches this value after only seven iterations.

Exponential functions (i.e. powers of e) may also be calculated by infinite series, the basic exponential function e^x being equivalent to the following:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \cdots + \frac{x^\infty}{\infty!}$$

Certain trigonometric functions are also infinite series of factorials. For example, the *cosine* and *sine* functions (with angle x expressed in units of *radians* rather than degrees) are equivalent to the following series:

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Shown here are partial sum expansions up to $n = 5$ for both cosine and sine functions:

$$\cos x \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!}$$

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!}$$

If we closely examine these two series and compare them with the exponential series (e^x, shown below), some similarities should become apparent. Pay particular attention to the denominators of the fractional terms:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \cdots + \frac{x^\infty}{\infty!}$$

Notice how the exponential series contains both odd- and even-numbered factorial terms, whereas the cosine series contains only even-numbered factorials and the sine series only odd-numbered factorials in their denominators. Note also how the cosine series begins with the term 1 while the sine series begins with x, and how the exponential series contains both 1 and x. Euler noticed these patterns too, and reasoned that the exponential series was some *combination* of the cosine and sine series. The major incongruity between the cosine, sine, and exponential series is the *signs* of the terms: the cosine and sine functions contain terms of alternating sign (i.e. positive, then negative, then positive, etc.) whereas the exponential series contains only positive terms.

Euler's stroke of genius was in recognizing that *imaginary numbers* could be used to harmonize the signs of these terms, and create an equivalence between the exponential series and the cosine/sine series. Recall that an "imaginary" number is found by taking the square root of a negative number, with i or j used to express $\sqrt{-1}$. If $j = \sqrt{-1}$, then $j^2 = -1$ and $j^3 = -j$ and $j^4 = 1$, etc.

Euler substituted jx for x in the exponential series, and made j a multiplying coefficient in the sine series, to show that the exponential series could be made the sum of the cosine and sine series. First, we will see what happens when we substitute jx for x in the exponential series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \cdots$$

$$e^{jx} = 1 + jx + \frac{(jx)^2}{2!} + \frac{(jx)^3}{3!} + \frac{(jx)^4}{4!} + \frac{(jx)^5}{5!} + \frac{(jx)^6}{6!} + \frac{(jx)^7}{7!} + \cdots$$

$$e^{jx} = 1 + jx - \frac{x^2}{2!} - j\frac{x^3}{3!} + \frac{x^4}{4!} + j\frac{x^5}{5!} - \frac{x^6}{6!} - j\frac{x^7}{7!} + \cdots$$

Look closely at what happens to the j terms in the middle equation as they become raised to powers of 2, 3, 4, 5, and beyond. Recall that $j^2 = -1$ and that $j^3 = -j$ and that $j^4 = 1$. Going beyond this, we see how $j^5 = j$ and $j^6 = -1$ and $j^7 = -j$ and $j^8 = 1$, etc.

Next, we will see what happens when we apply j as a multiplying coefficient in the sine series. This simply multiplies every term by j:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$j\sin x = jx - j\frac{x^3}{3!} + j\frac{x^5}{5!} - j\frac{x^7}{7!} + \cdots$$

Finally, when we set these three equations beside each other, it is apparent how e^{jx} is the sum of $\cos x$ and $j\sin x$. This fact may be clearly shown by using different colors (blue and red) to represent the cosine and sine terms, respectively, and writing the exponential series as an alternating series of those blue and red terms taken from the cosine and sine series:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

$$j\sin x = jx - j\frac{x^3}{3!} + j\frac{x^5}{5!} - j\frac{x^7}{7!} + \cdots$$

$$e^{jx} = 1 + jx - \frac{x^2}{2!} - j\frac{x^3}{3!} + \frac{x^4}{4!} + j\frac{x^5}{5!} - \frac{x^6}{6!} - j\frac{x^7}{7!} + \cdots$$

And with that, we have Euler's Relation:

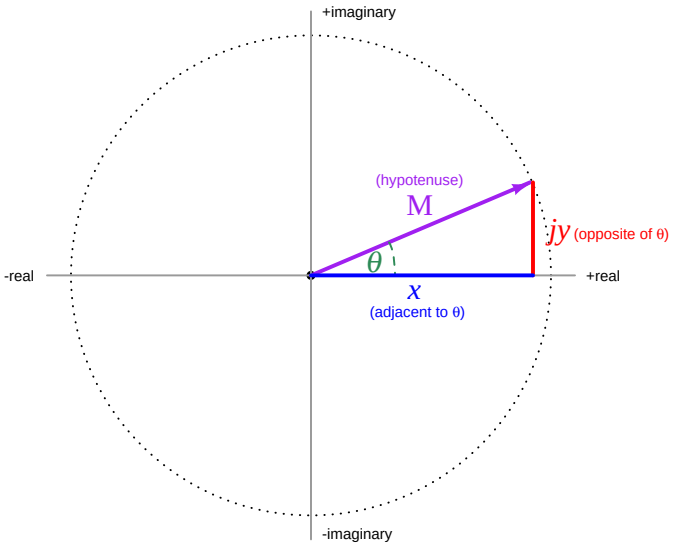
$$e^{jx} = \cos x + j\sin x$$

Euler's Relation is incredibly important in many different branches of engineering and science because it provides an equivalence between exponential functions which are relatively easy to manipulate using calculus, and trigonometric functions which are relatively difficult manipulate using calculus. Therefore, any application requiring the application of calculus to sinusoidal functions can benefit from Euler's Relation.

4.2 Complex-number arithmetic

Complex numbers are very useful in AC circuit analysis because each one has the ability to represent both a magnitude and a phase shift between that quantity and some other reference quantity. Despite the existence of electronic calculators and computer software capable of performing arithmetic on complex-number quantities, there are still times when we must perform some calculation on these quantities "by hand". This technical reference reviews the basic arithmetic operations on complex numbers, complete with examples.

Recall that complex numbers may be represented in either *rectangular* or *polar* form, rectangular being a quantity with both a "real" and an "imaginary" component, and polar being a quantity with a magnitude and an angle. Graphically, these two forms relate to the sides of a right triangle:



Rectangular form: $x + jy$ (where $j = \sqrt{-1}$)

Polar form: $M\angle\theta$

To convert from rectangular form to polar form, $M = \sqrt{x^2 + y^2}$ and $\theta = \arctan \frac{y}{x}$

To convert from polar form to rectangular form, $x = M \cos\theta$ and $y = M \sin\theta$

As we will see, addition and subtraction is easiest to do with rectangular-form notation while multiplication and division is easiest to do with polar-form notation. Thus, circuit analysis doing "long-hand" complex-number arithmetic often involves conversions back and forth between rectangular and polar forms in order to set up the quantities before applying Ohm's Law, Kirchhoff's Laws, etc. This can be tedious, and it is also prone to rounding errors. The reader is advised to store all intermediate results in their calculator's memory and recall when needed, rather than re-type quantities and thereby incur rounding errors due to truncation.

4.2.1 Negating complex numbers

The sign of a complex number may be reversed just as easily in rectangular form as in polar form. Rectangular-form negation consists of multiplying -1 through to both the real and imaginary terms. Polar-form negation consists solely of adding 180 degrees to the angle, or alternatively, by reversing the sign of the magnitude and leaving the angle alone.

Example: reverse the sign of $5 - j4$

$$-(5 - j4)$$

$$-5 + j4$$

Example: reverse the sign of $6\angle 30^o$

$$-(6\angle 30^o)$$

$$6\angle 210^o = 6\angle -150^o = -6\angle 30^o$$

4.2.2 Adding complex numbers

Complex numbers are most easily added in *rectangular form*: simply add the real portions and then add the imaginary portions.

Example: add $5 - j4$ to $-1 - j3$

$$(5 - j4) + (-1 - j3)$$

$$(5 + (-1)) + (-j4 + (-j3))$$

$$4 - j7$$

4.2.3 Subtracting complex numbers

Complex numbers are most easily subtracted in *rectangular form*: simply subtract the real portions and then subtract the imaginary portions.

Example: subtract $5 - j4$ from $-1 - j3$

$$(-1 - j3) - (5 - j4)$$

$$(-1 - (5)) + (-j3 - (-j4))$$

$$-6 + j1$$

4.2.4 Multiplying complex numbers

Complex numbers are most easily multiplied in *polar form*: simply multiply the magnitudes and add the angles.

Example: multiply $6\angle 30^o$ by $2\angle -10^o$

$$(6\angle 30^o) \times (2\angle -10^o)$$

$$(6 \times 2)\angle(30^o + (-10^o))$$

$$12\angle 20^o$$

Multiplication of rectangular-form complex numbers less straight-forward then with polar-form numbers, and resembles multiplication of algebraic polynomials:

Example: multiply $5 - j4$ by $-1 - j3$

$$(5 - j4) \times (-1 - j3)$$

$$(5 \times (-1)) + (5 \times (-j3)) + (-j4 \times (-1)) + (-j4 \times (-j3))$$

$$(-5) + (-j15) + (j4) + (j^2 12)$$

$$(-5) + (-j15) + (j4) + ((-1)12)$$

$$(-5) + (-j15) + (j4) + (-12)$$

$$-17 - j11$$

4.2.5 Dividing complex numbers

Complex numbers are most easily divided in *polar form*: simply divide the magnitudes and subtract the angles.

Example: divide $6\angle 30^o$ by $2\angle -10^o$

$$\frac{6\angle 30^o}{2\angle -10^o}$$

$$\frac{6}{2}\angle(30^o - (-10^o))$$

$$3\angle 40^o$$

4.2.6 Reciprocating complex numbers

Reciprocation is division into one, and so complex numbers are reciprocated most easily in *polar form* just as division is best performed in polar form: simply reciprocate the magnitude and negate the angle.

Example: reciprocate $2\angle -10^o$

$$\frac{1}{2\angle -10^o}$$

$$\frac{1}{2}\angle -(-10^o)$$

$$0.5\angle 10^o$$

4.2.7 Calculator tips

Here is some advice when using calculators to do complex-number arithmetic:

- When manually entering a complex-number value, enclose that value in parentheses. Some calculators struggle to properly perform order-of-operations with complex numbers. For example, some calculators will interpret $45\angle 30^o \times 5$ as $45\angle(30^o \times 5)$ to give $45\angle 150^o$ when what was really intended was $(45\angle 30^o) \times 5 = 225\angle 30^o$. Also, note that the practice of highlighting previous results in a multi-line display and then "pasting" those results into a new calculation may suffer similar problems.
- *Never* re-enter a non-round computed result, but instead save that to a memory location and then recall from memory when needed for further calculations. You will find that rounding errors compound *aggressively* in complex-number arithmetic, and so the general good habit of using memory locations becomes a near-necessity with these calculations. Another important benefit to using memory locations is the avoidance of the order-of-operations problem mentioned previously: when recalling a complex-number value from memory and then placing that variable name (e.g. x) into subsequent calculations, the calculator treats the memory variable as a complete number rather than incorrectly operating on only one of its parts.

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`.

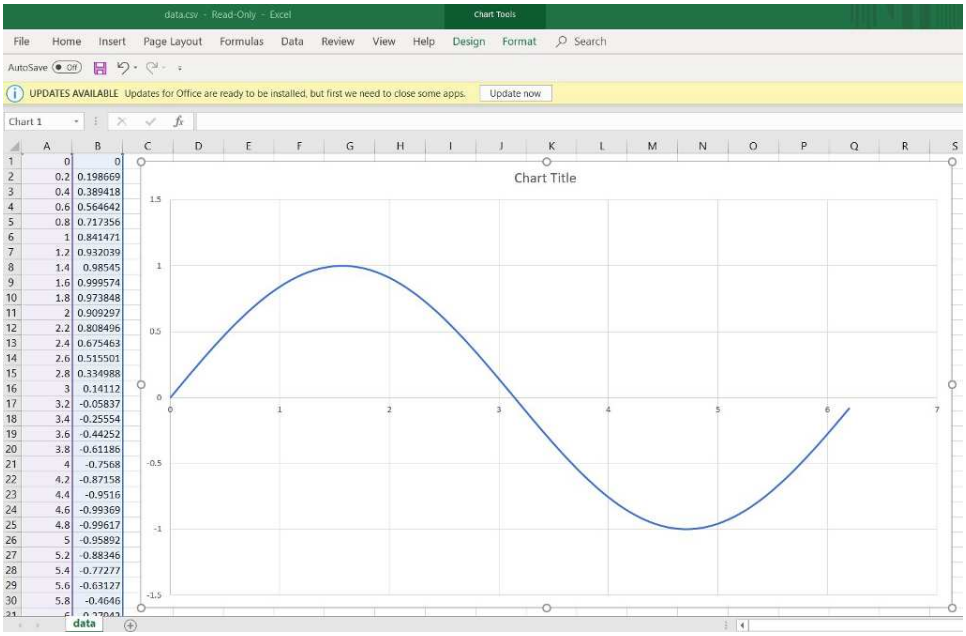
As previously mentioned, C++ lacks a standard library of graphics functions for plotting curves and other mathematical shapes to the computer's screen, which is why we opted to use *standard console* characters to do the same. If a truly *graphic* output is desired for our waveform plot, there are relatively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

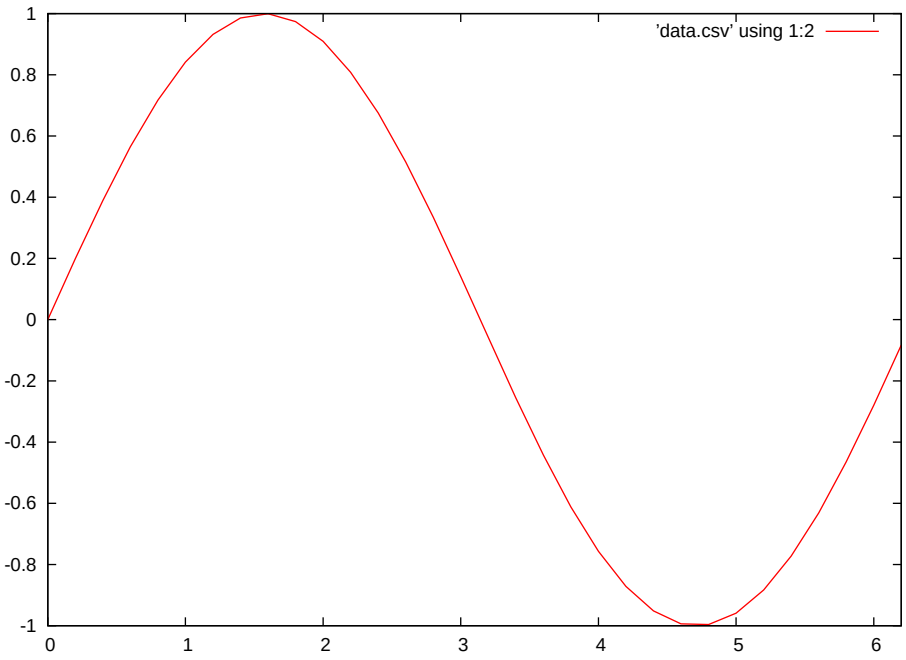


[11] A relatively easy way to do this is to run the C++ program from a console, using the *redirection* symbol (`>`). For example, if we saved our source code file under the name `sinewave.cpp` and then entered `g++ -o sinewave.exe sinewave.cpp` at the command-line interface to compile it, the resulting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

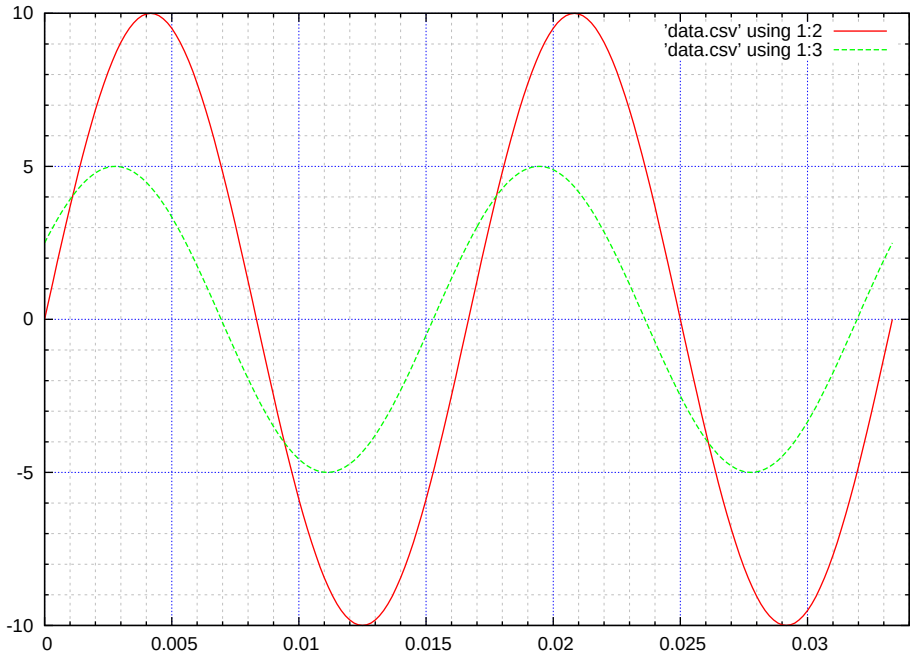
The sine functions are computed within these last `cout` instructions. The product of time and frequency (*seconds* times *cycles per second*) yields a result in *cycles*. Phase shift was entered in *degrees*, so division by 360 is necessary to case phase shift into cycles because there are 360 degrees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

The following C++ program accepts numerical values entered by the user, stores them in a set of variables called an *array*, and then plots them to the console. This is known as an *arbitrary waveform* algorithm because it allows the user to define a waveform of any shape simply by specifying its amplitude point by point:

```
#include <iostream>
using namespace std;
#define MAX 10

int plot(float);
float wave[MAX];

int main (void)
{
  int phase;

  for (phase = 0 ; phase < MAX ; ++phase)
  {
    cout << "Enter instantaneous value for phase index " << phase << ":" ;
    cin >> wave[phase];
  }

  while (1)
    for (phase = 0 ; phase < MAX ; ++phase)
      plot(wave[phase]);

  return 0;
}

int plot(float x)
{
  int n;

  for (n = 0 ; n < (3 * x) ; ++n)
    cout << " ";

  cout << "*" << endl;

  return 0;
}
```

In the following example, we run this program and enter at the prompt values which will form a crude square wave when plotted. After entering the tenth value the program proceeds to endlessly plot this waveform (and in fact must be forcibly interrupted by the <Ctrl-C> key combination:

```
Enter instantaneous value for phase index 0:10
Enter instantaneous value for phase index 1:15
Enter instantaneous value for phase index 2:15
Enter instantaneous value for phase index 3:15
Enter instantaneous value for phase index 4:15
Enter instantaneous value for phase index 5:10
Enter instantaneous value for phase index 6:5
Enter instantaneous value for phase index 7:5
Enter instantaneous value for phase index 8:5
Enter instantaneous value for phase index 9:5
                            *
                                          *
                                          *
                                          *
                                          *
                            *
                  *
                  *
                  *
                  *
                            *
                                          *
                                          *
                                          *
                                          *
                            *
                  *
                  *
                  *
                  *
```

We will now explore the operation of this program, discussing the following programming principles as we proceed:

- Preprocessor directives (`#include`, `#define` macros), namespaces
- Custom functions: prototyping, return values, arguments
- The `main` function: return values, arguments
- Delimiter characters (e.g. { } ;)
- Variable types (`float`, `int`, `bool`), names, and declarations

- Arrays
- Accepting user input (`cin`, `>>`)
- Loops (`while`, `for`)
- Increment/decrement operators (`++`, `--`)
- Nested loops
- Printing text output (`cout`, `<<`, `endl`)

At the top of the source code listing are several lines of code instructing the compiler software how to interpret what follows. The `#include` directive tells the compiler to read the contents of the `iostream` file included in its library of files, and those contents define some of the instructions used later in our program. `namespace` instructs the compiler to interpret commands according to their *standard* (`std`) definitions.

The `#define` directive is known in computer programming as a *macro*, and it is substitutionary in nature: in this case it instructs the compiler to substitute `10` for any instance of `MAX` found in the rest of the code. In this program, the function of `MAX` is to establish the number of data points constituting our arbitrary waveform. Such directives are useful when we wish to easily modify some key value or word utilized multiple times in our code, simply by editing a single line defining that key value or word.

Following the `#define` directive is the line of code `int plot(float)` which *prototypes* a custom function we are calling `plot`. *Functions* in C and C++ are selected lines of code which may be "called" to execute by other lines of code within the program, and in so doing the regular top-to-bottom flow of execution "jumps" to the function, executes it, and then "returns" where it left off. Some programming languages call these selected portions of code *subroutines*, but in C/C++ they are called *functions*. The "prototype" line simply alerts the compiler that a function by the name of `plot` will occur in this program, that it will accept a single floating-point (`float`) variable for an "argument", and that it will return an integer (`int`) variable when finished.

The actual code comprising the `plot` function appears at the very bottom of the listing. The `int plot(float x)` labels this function and names its argument value `x`. The two brace symbols ({ }) surround the lines of code belonging to the `plot` function.

Looking back toward the top of the listing, we see another function called `main`. Like `plot`, it has a pair of curly-brace characters preceding and following its collection of code lines, defining for the compiler which lines of code belong to `main`. The `main` function accepts no input values, but like the `plot` function it does return the integer value zero.

The next line down from the top is a *declaration* reserving space in the computer's memory to store variable data. In fact, there are multiple declarations in this program: `int phase`, `int n`, and the one farthest toward the beginning, `float array[MAX]`. `int` refers to an integer number, while `float` refers to a floating-point number. The most distinctive part of the `float` declaration, however, is not the fact that it is for a floating-point variable, but rather that it is declaring an *array of ten* floating-point variables, each one named `wave` and having a subscript value. This is analogous to a mathematical expression using multiple variables of the same letter-name and differentiated only

by subscript (e.g. R_1, R_2, R_3, etc.). In this case, the ten floating-point variables in this array are named `wave[0]`, `wave[1]`, `wave[2]`, etc. through `wave[9]`.

Arrays are very useful in C/C++ programming because they allow a set of variables to be addressed by a subscript or index number. This program uses the ten-element `wave` array to store amplitude values for ten points within the waveform's period.

Next in the `main` function is a `for` loop, the purpose of which being to repeatedly execute some line or lines of code according to the value of a variable. In this first `for` loop we prompt the user to enter instantaneous values for the waveform at ten different points[14] along its period, the variable `phase` incrementing from a starting value of zero (`phase = 0`) in steps of one (`++phase`) so long as its value is less than the maximum allowed (`phase < MAX`).

Here it is clear to see the utility of using an *array* of floating-point variables to store these waveform values as opposed to, say, ten differently-named floating-point variables. The exact same line of code accepting typed values from the user (`cin >> wave[phase]`) is re-used over and over again, a feat made possible by the fact that the integer variable `phase` has a new value each time, and therefore `cin` places the user's entry into a new element of the array each time.

One more note regarding the `for` loop is its use of brace symbols to encapsulate multiple lines of code to be repeatedly executed. Bear in mind that this is only necessary when *multiple* lines of code are involved. If just a single line of code will be "looped" the brace symbols become unnecessary, the single line merely has to immediately follow[15] the `for` statement.

After the first `for` loop we have a set of *nested loops* consisting of another `for` loop inside of a `while` loop. A `for` loop, as we know, repeats only as long as its variable remains within the specified bounds. A `while` loop, by contrast, repeats so long as its argument (the statement within the parentheses) is *logically true*. Since our `while` loop argument is one (which is by definition "true" in Boolean logic), it loops forever. Within that `while` loop is a `for` loop with the exact same arguments used in the previous `for` loop, serving a nearly identical purpose: incrementing the `phase` variable from zero to one less than `MAX` to address each element of the `wave` array in sequential order.

The solitary line of code executed by these nested loops is a *function call* to our custom `plot` function. With each iteration, `plot(wave[phase])` sends the value stored in array element `wave[phase]` to the `plot` function as an argument. Execution flow then jumps down to the `plot` function where this argument is renamed `x` and used within a `for` loop there to print a proportionate number of blank spaces to the computer's console display using the `cout` statement. At the end of the printed series of blank spaces, the last `cout` statement prints a star character (`*`). When finished with the `plot` function, execution flow returns where it left off. As the `plot` function gets repeatedly called, we end up with a repeated pattern of star characters on the console display forming the outline of the waveform.

[14] Or phase angles, if we envision the waveform's period as being a full circle, with each of the ten points being increments of rotation about the circle's center.

[15] Remember that C/C++ ignores whitespace, and due to the lack of a semicolon at the end of the `for` line the compiler will "see" any immediately-following line of code as a continuation of the `for` line.

The following examples show other arbitrary waveforms specified and plotted by this program.

First, a triangle wave:

```
Enter instantaneous value for phase index 0:10
Enter instantaneous value for phase index 1:11
Enter instantaneous value for phase index 2:12
Enter instantaneous value for phase index 3:13
Enter instantaneous value for phase index 4:14
Enter instantaneous value for phase index 5:15
Enter instantaneous value for phase index 6:14
Enter instantaneous value for phase index 7:13
Enter instantaneous value for phase index 8:12
Enter instantaneous value for phase index 9:11
                               *
                                  *
                                     *
                                        *
                                           *
                                              *
                                           *
                                        *
                                     *
                                  *
                               *
                                  *
                                     *
                                        *
                                           *
                                              *
                                           *
                                        *
                                     *
                                  *
                               *
```

Next, a sawtooth wave:

```
Enter instantaneous value for phase index 0:10
Enter instantaneous value for phase index 1:11
Enter instantaneous value for phase index 2:12
Enter instantaneous value for phase index 3:13
Enter instantaneous value for phase index 4:14
Enter instantaneous value for phase index 5:15
Enter instantaneous value for phase index 6:16
Enter instantaneous value for phase index 7:17
Enter instantaneous value for phase index 8:18
Enter instantaneous value for phase index 9:19
                               *
                                  *
                                     *
                                        *
                                           *
                                              *
                                                 *
                                                    *
                                                       *
                                  *
                                     *
                                        *
                                           *
                                              *
                                                 *
                                                    *
                                                       *
```

Next, an exponential-ramp wave:

```
Enter instantaneous value for phase index 0:0.0312
Enter instantaneous value for phase index 1:0.0625
Enter instantaneous value for phase index 2:0.125
Enter instantaneous value for phase index 3:0.25
Enter instantaneous value for phase index 4:0.5
Enter instantaneous value for phase index 5:1
Enter instantaneous value for phase index 6:2
Enter instantaneous value for phase index 7:4
Enter instantaneous value for phase index 8:8
Enter instantaneous value for phase index 9:16
 *
 *
 *
 *
  *
   *
      *
         *
            *
                        *
 *
 *
 *
 *
  *
   *
      *
         *
            *
                        *
```

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 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

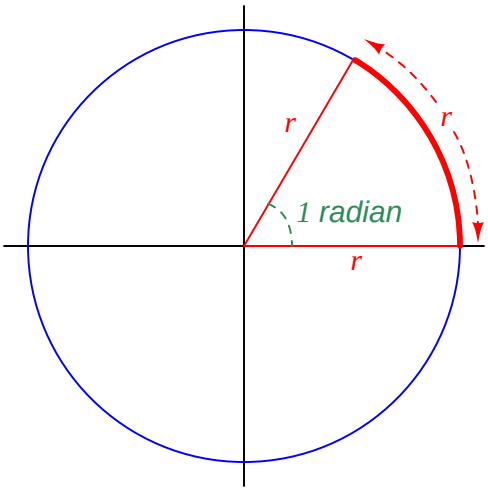
Phasor

Phase angle

Complex numbers

6.1.3 Radians

A *radian* is defined as that angle describing a sector of a circle, whose arc length is equal to the radius of the circle:



Based on this definition, how many radians comprise a full circle?

How many degrees are in one radian?

Radians are considered a "natural" measurement unit for angles, while degrees are "arbitrary". Explain why this is.

Challenges

- Calculate the angular velocity of a waveform having a frequency of 60 Hz (i.e. 60 cycles, or revolutions, per second).

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 Practice: complex number calculations

These complex-number arithmetic problems are presented to you, complete with answers (shown in **bold**), for the purpose of practice, since nearly all AC circuit calculations will need to be performed using complex numbers. Use these practice calculations to check your ability either to perform these calculations "by hand" (using trigonometric functions) or your ability to use your calculator's complex-number functionality.

Note: electronic hand calculators and computer-based calculation programs use the proper mathematical notation i to represent imaginary numbers rather than j. The letter j is used in electrical engineering work in order to avoid confusion with i being misinterpreted as "current". Also note that calculators and software programs usually default to *radians* for angle measurement rather than *degrees*, and will have to be configured (or converted) for degrees in order to handle the polar-form complex quantities shown here. Check the "mode" options of your hand calculator to ensure angles are in the correct unit and also that it will display in either rectangular or polar.

Addition and subtraction:

$(5 + j6) + (2 - j1) = \mathbf{7 + j5}$
$(10 - j8) + (4 - j3) = \mathbf{14 - j11}$
$(-3 + j0) + (9 - j12) = \mathbf{6 - j12}$

$(3 + j5) - (0 - j9) = \mathbf{3 + j14}$
$(25 - j84) - (4 - j3) = \mathbf{21 - j81}$
$(-1500 + j40) + (299 - j128) = \mathbf{-1201 - j88}$

$(25\angle 15^o) + (10\angle 74^o) = \mathbf{31.35\angle 30.87^o}$
$(1000\angle 43^o) + (1200\angle -20^o) = \mathbf{1878.7\angle 8.311^o}$
$(522\angle 71^o) - (85\angle 30^o) = \mathbf{461.23\angle 77.94^o}$

Multiplication and division:

$(25\angle 15^o) \times (12\angle 10^o) = \mathbf{300\angle 25^o}$
$(1\angle 25^o) \times (500\angle -30^o) = \mathbf{500\angle -5^o}$
$(522\angle 71^o) \times (33\angle 9^o) = \mathbf{17226\angle 80^o}$

$\frac{10\angle -80^o}{1\angle 0^o} = \mathbf{10\angle -80^o}$

$\frac{25\angle 120^o}{3.5\angle -55^o} = \mathbf{7.142\angle 175^o}$

$\frac{-66\angle 67^o}{8\angle -42^o} = \mathbf{8.25\angle -71^o}$

$(3 + j5) \times (2 - j1) = \mathbf{11 + j7}$
$(10 - j8) \times (4 - j3) = \mathbf{16 - j62}$

$\frac{(3+j4)}{(12-j2)} = \mathbf{0.1892 + j0.3649}$

Reciprocation:

$$\frac{1}{(15\angle 60^o)} = \mathbf{0.0667\angle -60^o}$$

$$\frac{1}{(750\angle -38^o)} = \mathbf{0.00133\angle 38^o}$$

$$\frac{1}{(10+j3)} = \mathbf{0.0917 - j0.0275}$$

$$\frac{1}{\frac{1}{15\angle 45^o}+\frac{1}{92\angle -25^o}} = \mathbf{14.06\angle 36.74^o}$$

$$\frac{1}{\frac{1}{1200\angle 73^o}+\frac{1}{574\angle 21^o}} = \mathbf{425.7\angle 37.23^o}$$

$$\frac{1}{\frac{1}{23k\angle -67^o}+\frac{1}{10k\angle -81^o}} = \mathbf{7.013k\angle -76.77^o}$$

$$\frac{1}{\frac{1}{110\angle -34^o}+\frac{1}{80\angle 19^o}+\frac{1}{70\angle 10}} = \mathbf{29.89\angle 2.513^o}$$

$$\frac{1}{\frac{1}{89k\angle -5^o}+\frac{1}{15k\angle 33^o}+\frac{1}{9.35k\angle 45}} = \mathbf{5.531k\angle 37.86^o}$$

$$\frac{1}{\frac{1}{512\angle 34^o}+\frac{1}{1k\angle -25^o}+\frac{1}{942\angle -20}+\frac{1}{2.2k\angle 44^o}} = \mathbf{256.4\angle 9.181^o}$$

Sign reversal:

$-(45\angle 70^o) = \mathbf{45\angle -110^o}$

$-(90\angle -20^o) = \mathbf{90\angle 160^o}$

$-(5 + j8) = \mathbf{-5 - j8}$

$-(-3 + j9) = \mathbf{3 - j9}$

$-(10 - j15) = \mathbf{-10 + j15}$

Practical suggestions for using your calculator to perform these operations:

- Surround each complex-number quantity with parentheses when setting up arithmetic operations; e.g., `(3 + j5) * (4 - j2)` instead of `3 + j5 * 4 - j2`. This habit will guarantee your calculator executes the desired order of operations rather than assert its own. For instance, in the example given here the calculator may choose to multiply $j5$ by 4 and then add on 3 and $-j2$ since multiplication typically precedes addition, if the two complex numbers are not encapsulated in their own sets of parentheses.

- Store all calculated results in memory and then recall from memory when re-using those values, rather than re-entering previously-calculated values by hand or sampling previously-calculated values from the multi-line display. Manually re-entering values invites rounding errors and keystroke errors in all cases, and I've found certain calculators (I'm looking at you, *TI*!) fail to properly enter complex-number values when sampled from their multi-line displays. Getting in the habit of using your calculator's memory locations is an all-around good habit that will serve you very well!

Some Texas Instruments brand calculators such as the TI-84 offer an exponential key and imaginary (i) key which allows you to enter numbers in complex exponential form (i.e. $e^{i\theta}$). With the TI-84, for example, the complex number $10 - j8$ may be entered in either of the two following forms:

`(10 - i8)` or `(10 - 8i)`

The result may be displayed in either rectangular or polar forms according to the complex-number display *mode* the TI-84 calculator has been set to. In rectangular mode the displayed result for $10 - j8$ will be `10 - 8i`, whereas in polar mode the displayed result will be `12.806` $\mathtt{e}^{-38.66i}$. Note how the TI-84 uses exponential notation for polar display, where the angle (-38.66 degrees, in this example) is an imaginary power of e.

If you wish to enter a complex number in polar form on a TI-84, you must unfortunately express the angle in units of *radians* (even though the calculator is able to display the result in *degrees*). For example, to enter the number $25\angle 15^o$ into a TI-84 calculator, you must type:

`25` $\mathtt{e}^{i15\pi/180}$

The fraction $\pi/180$ is the conversion factor from degrees to radians, since there are 2π radians to a full circle, or π radians to every 180 degrees. Thus, writing $15\pi/180$ multiplies the desired angle (15 degrees) by the conversion factor $\pi/180$ to yield a power in radians. The obligatory i simply makes this power an imaginary quantity, which is mathematically necessary with exponential notation for describing a complex number. It should be noted that the order of entry for the power matters little. $i15\pi/180$ works just as well as $15i\pi/180$ or $15\pi/180i$.

A time-saving step some students find useful is to save the imaginary quantity $i\pi/180$ to a memory location in the TI-84 such as Z. That way, they can recall that imaginary factor from memory instead of typing the whole thing by hand every time they wish to enter a polar-form complex number. Supposing the memory location Z contains $i\pi/180$, entering the number $25\angle 15^o$ becomes as simple as:

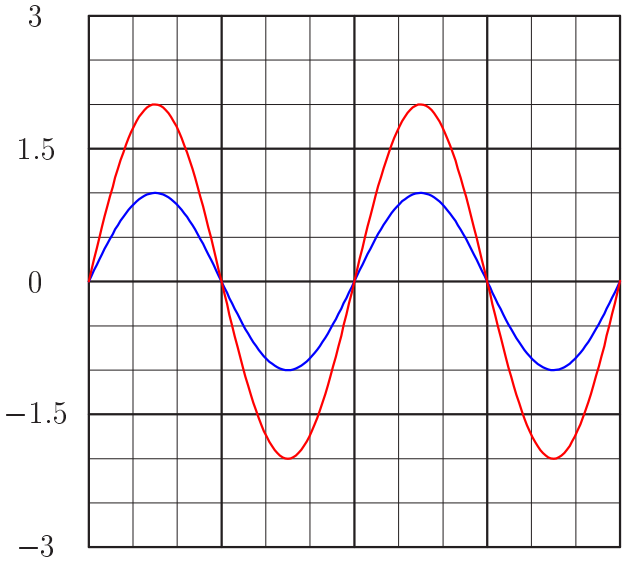
25 e^{15Z} or 25 e^{Z15}

It should be understood that *any* memory location in your calculator is suitable for storing $i\pi/180$, not just Z. The TI-84 calculator even provides a Θ memory location (<Alpha>-<3>) that you may use and find easy to remember because of its common association with angles. It should also be understood that this imaginary quantity is not the same as i or j, which the calculator already provides a dedicated function for. The imaginary quantity we're storing in memory for the purpose of entering polar-notation angles contains not only i but also the $\pi/180$ conversion factor necessary for translating your *degree* entry into *radians*.

Some calculators provide easier means of entering and displaying complex numbers in polar form. Both the Texas Instruments TI-36X Pro and TI-89 offer an angle symbol ($\angle$) for this purpose, the TI-36X Pro being a much less expensive and less complex calculator than the TI-89. Entering the number $25 \angle 15^o$ into one of these calculators is as easy as typing 25 $\angle$ 15, and likewise the result will be displayed in this same form when the calculator is set to "polar" complex mode.

6.2.3 Adding sine waves

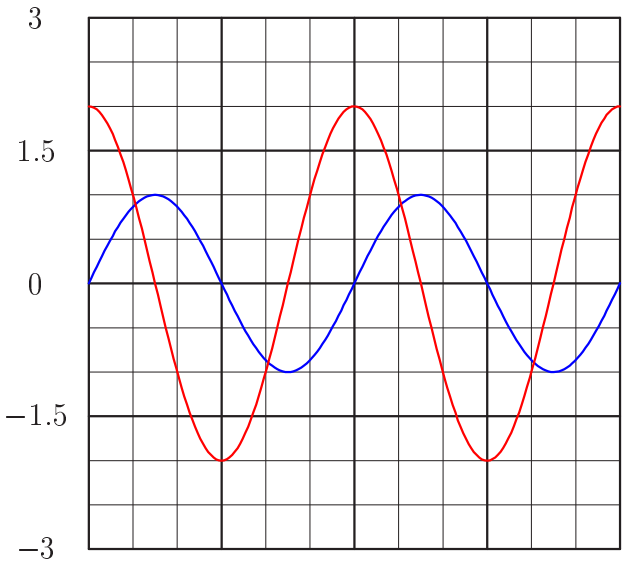
Using a computer or graphing calculator, plot the sum of these two sine waves:



What do you suppose the sum of these two sine waves will look like, seeing that they are perfectly in-phase with each other? Hint: you will need to execute equations within a programming "loop" that look something like this:

- `y1 = sin(x)`
- `y2 = 2 * sin(x)`
- `y3 = y1 + y2`

Now do the same with these two sine waves:



What do you suppose the sum of these two sine waves will look like, seeing that they are out of phase with each other? Hint: you will need to execute equations within a programming "loop" that look something like this:

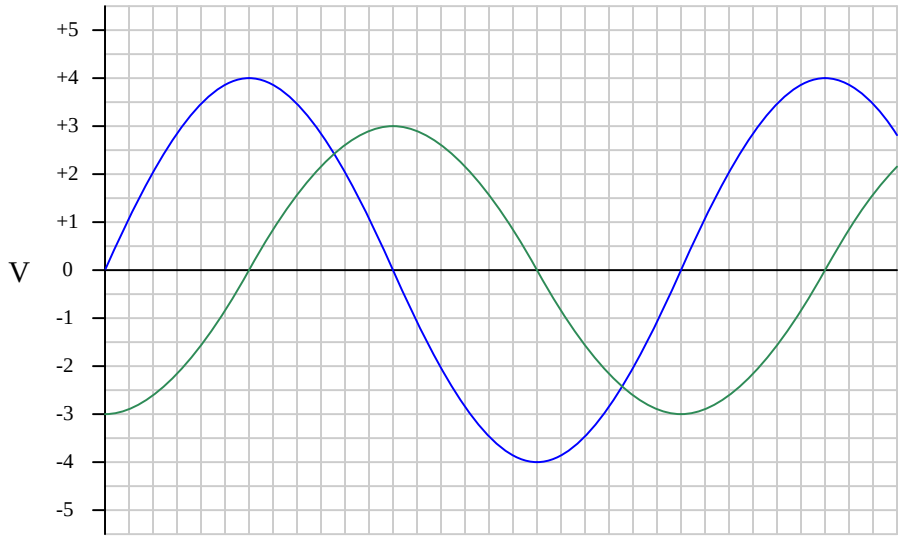
- `y1 = sin(x)`
- `y2 = 2 * sin(x + 90`^{o}`)`
- `y3 = y1 + y2`

Challenges

- Many electronic calculators and computer programming languages assume trigonometric functions require angle values in *radians* rather than *degrees*. Re-write each of the sine calculation functions where a value of x in degrees gets converted into radians prior to executing the sine function.

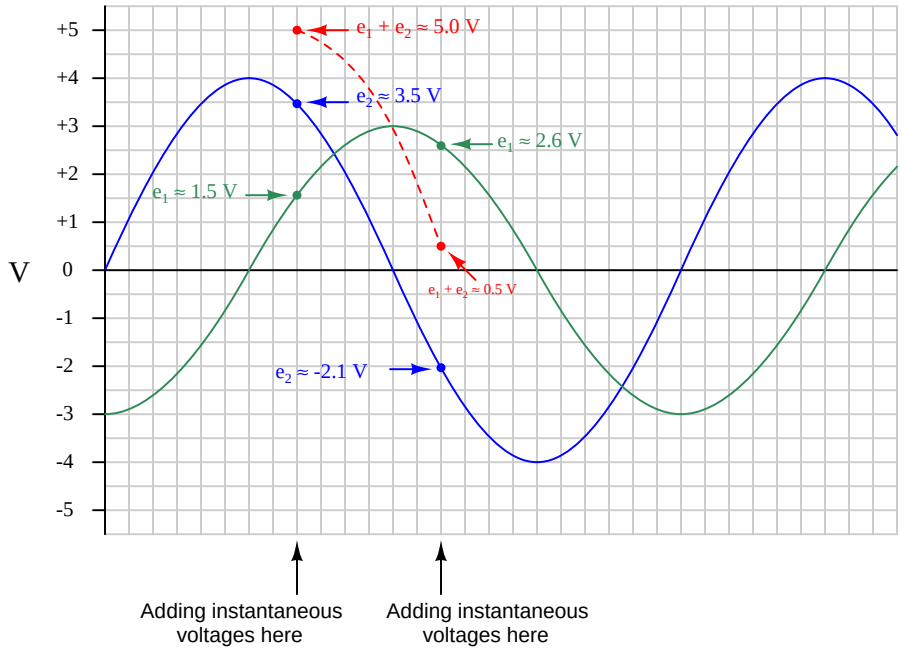
6.2.4 Hand-plotting a sinusoidal sum

Shown here are two sine waves of equal frequency, superimposed on the same graph:



As accurately as possible, determine the amount of phase shift between the two waves, based on the divisions shown on the graph.

Also, plot a third sine wave that is the *sum* of the two sine waves shown based on points found on each of the two sinusoidal curves. Again, do this as accurately as possible, based on the divisions shown on the graph. To give an example of how you might do this, observe the following illustration:



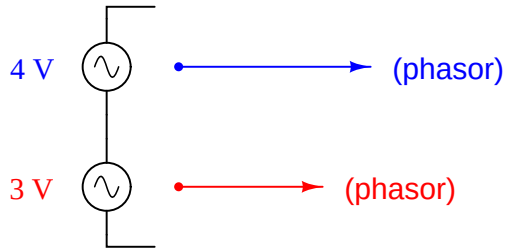
Calculate the peak value of the resultant (sum) sine-wave, and compare this with the peak values of the two original sine waves.

Challenges

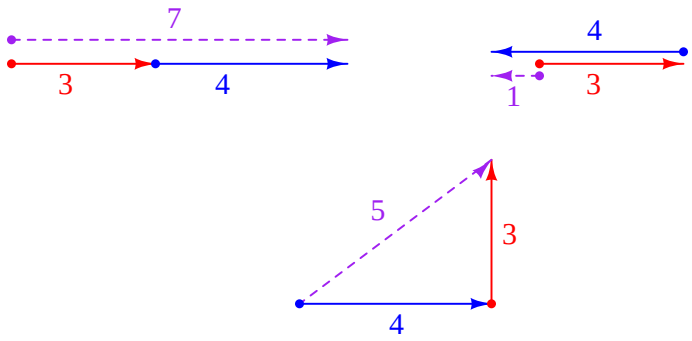
- Sketch a right-triangle with side-lengths equal to the peak amplitudes of these three sine waves (the two given voltage waveforms and their sum).

6.2.5 Phasor addition of two AC voltages

Special types of vectors called *phasors* are often used to depict the magnitude and phase-shifts of sinusoidal AC voltages and currents. Suppose that the following phasors represent the series summation of two AC voltages, one with a magnitude of 3 Volts and the other with a magnitude of 4 Volts:



Explain what each of the following phasor diagrams represents, in electrical terms:



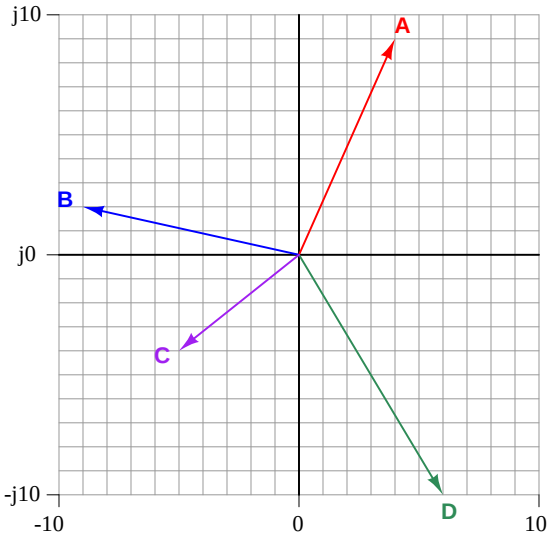
Also explain the significance of these sums: that we may obtain three *different* values of total voltage (7 Volts, 1 volt, or 5 Volts) from the same series-connected AC voltages. What does this mean for us as we prepare to analyze AC circuits using the rules we learned for DC circuits?

Challenges

- In DC circuits, it is permissible to connect multiple voltage sources in parallel, so long as the voltages (magnitudes) and polarities are the same. Is this also true for AC? Why or why not?

6.2.6 Simple phasor diagrams

Write both the rectangular and polar expressions for all phasors shown in this diagram:



$A =$
$B =$
$C =$
$D =$

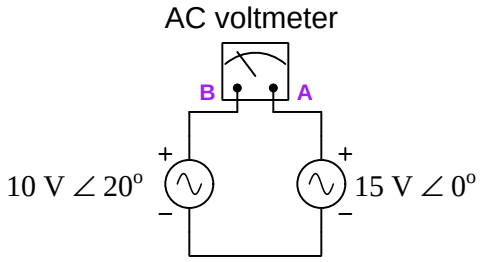
Next, explain how a phasor diagram might be a helpful method to check your work on certain types of complex-number calculations.

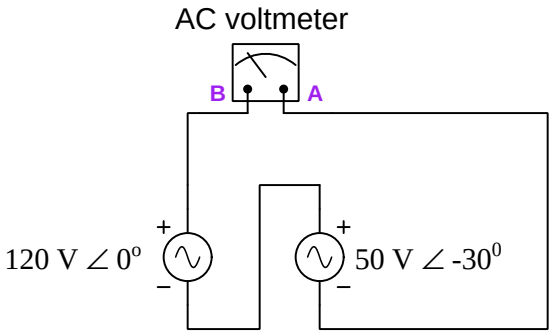
Challenges

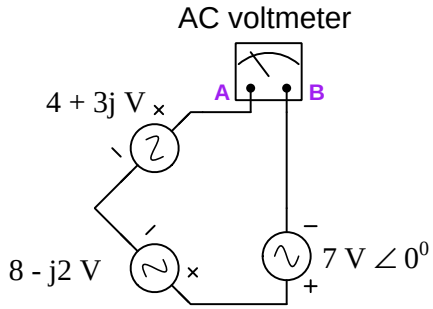
- How may we show the addition of two or more complex-number quantities on a phasor diagram?
- Which electrical quantities add in a series network?
- Which electrical quantities add in a parallel network?

6.2.7 Series AC voltages

Determine the voltage between test points A and B in each of these example circuits. First, treat all sources as DC instead of AC with DC voltage values equal to the AC (polar) magnitudes shown, and calculate V_{AB} as it would be registered by a DC voltmeter (with red lead on A and black lead on B). Then, treat all sources as AC all with the same frequency with the phase angles shown and calculate V_{AB}, expressing that total voltage in either polar or rectangular notation:







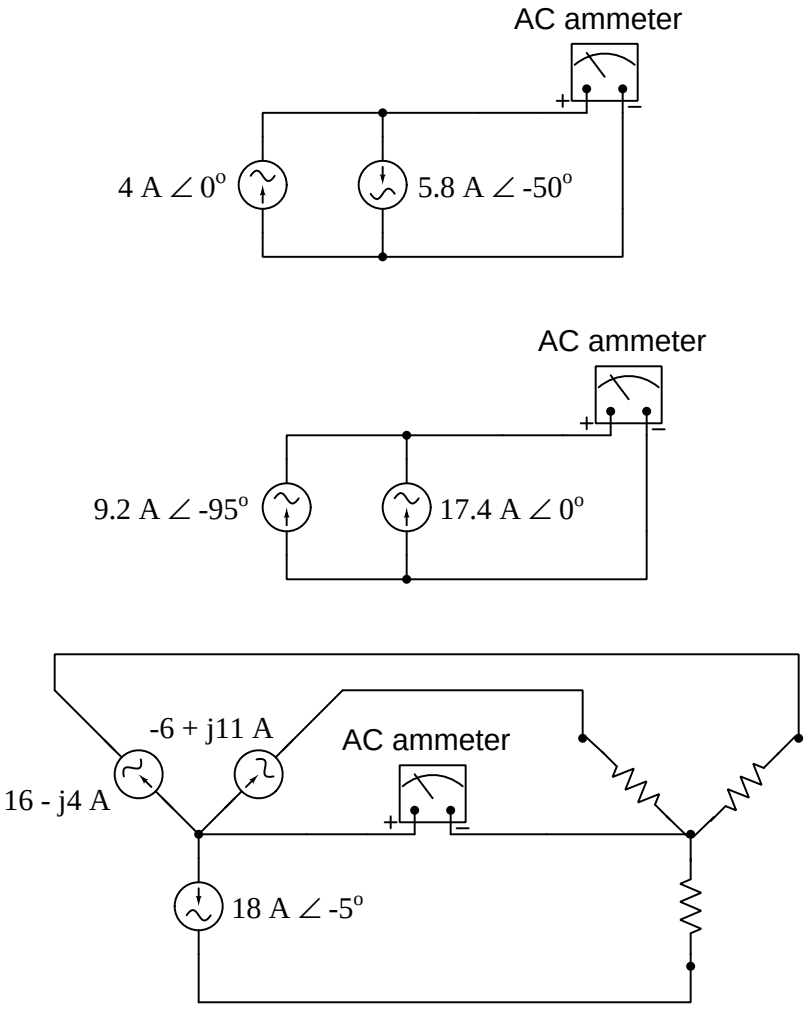
Challenges

- Suppose each of these sources was DC instead of AC (i.e. the polar magnitude of each became the DC voltage value). Compute V_{AB} given these new values

- How would each of these answers differ, if at all, if test points A and B were reversed?

6.2.8 Parallel AC currents

Determine the current registered by each ammeter in these example circuits. First, treat all sources as DC instead of AC with DC current values equal to the AC (polar) magnitudes shown, and calculate I_{meter} as it would be registered by a DC ammeter. Then, treat all sources as AC all with the same frequency with the phase angles shown and calculate I_{meter}, expressing that total current in either polar or rectangular notation:



Challenges

- How would each of these answers differ, if at all, if the ammeter's test leads were reversed?

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.

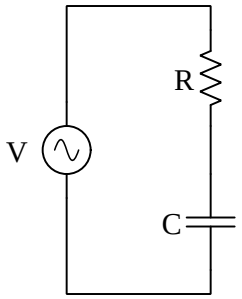
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of 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 Incorrect voltage calculation

A student learning about AC circuits for the first time constructs the following resistor-capacitor circuit in the lab:



Measuring 3.75 Volts AC across the resistor and 5.05 Volts across the capacitor, the student concludes the source's voltage must be 8.8 Volts AC. However, when they measure the source's voltage, their voltmeter only registers 6.29 Volts.

Identify the student's error, and explain how they should have calculated total (source) voltage.

Challenges

- What fundamental principle of electric circuits did the student attempt to apply in adding V_R and V_C? Is this principle still valid for AC circuits, or does it apply only to DC circuits?

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: casting new concepts in one's own words, running mental experiments to perceive patterns and trends, etc. 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. Depth of understanding and permanence of memory are both directly related to mental effort during the learning process – i.e. *easy learning is shallow and fleeting.*

- **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 technical 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 to dialogue with the industry representatives at those meetings.
- 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.
- Hold mandatory "check-in" meetings between all program faculty and each new student during their first term. Offer these to all other students as an option, except for any students continuing to manifest unprofessional behaviors, poor academic performance, or who have some other need for a face-to-face meeting with faculty.
- Address problems while they are small, before they grow larger. This is equally true for tutoring technical concepts as it is for 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

Steinmetz, Charles Proteus, "Complex Quantities and their use in Electrical Engineering", *Proceedings of the International Electrical Congress* held in Chicago (1893), edited by Max Osterberg, pages 33-75, American Institute of Electrical Engineers, 1894.

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.

30 August 2026 – added +1 and −1 labels to image_0405.

27 August 2025 – added more explanation on complex numbers to the Simplified Tutorial chapter, at the recommendation of Jason Durrett.

26-28 August 2024 – divided the Introduction chapter into sections, one with recommendations for students, one with a listing of challenging concepts, and one with recommendations for instructors. Also edited image_3152 to show dashed projection lines.

25 January 2024 – added practical suggestions for calculator usage to the end of the "Practice: complex number calculations" Quantitative Reasoning question.

13 September 2023 – added comments about scalar numbers into the Simplified and Full Tutorials. Also corrected a grammatical error, replacing the nonexistent word "incongruency" with *incongruity*.

30 August 2023 – clarified j versus i in the Simplified Tutorial chapter.

3 February 2023 – clarified confusion with "amplitude" versus "magnitude" which included editing graphics too (e.g. image_0412, image_0414, image_0416).

26 January 2023 – minor edits to the Full Tutorial, and some corrections to 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.

18 May 2022 – placed the "Practice: complex number calculations" questions into its own file (case_complexpractice.latex) so it may be shared amongst multiple modules.

3 February 2022 – added another part to the "Simple phasor diagrams" Quantitative Reasoning question. Also added a short amount of text to the Full Tutorial describing vector addition (for the 3-4-5 triangle).

23 December 2021 – minor addition to the Tutorial commenting on the use of cosine and sine functions to convert from polar into rectangular forms.

8 May 2021 – commented out or deleted empty chapters.

18 March 2021 – corrected multiple instances of "volts" that should have been capitalized "Volts".

28 January 2021 – divided the Full Tutorial chapter into separate sections.

10 January 2021 – minor additions to both the Simplified and Full Tutorial chapters commenting on + and − polarity marks applied to AC voltage sources.

29 October 2020 – added some Challenge questions.

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.

6 May 2020 – added reference to Charles Steinmetz's seminal paper on complex quantities in AC circuits.

20 April 2020 – added a another Programming Reference section using C++ to plot two sinusoidal waveforms with their own phase shifts.

26 March 2020 – added source voltage measurement to the "Incorrect voltage calculation" Diagnostic Reasoning problem.

25 March 2020 – added tips on calculator usage for the Practice problem on complex number calculations.

8 March 2020 – added some Quantitative Reasoning problems.

5 March 2020 – edited the Simplified Tutorial, adding the "longhand" voltage summation example at the very end after showing how a suitable hand calculator can do it automatically.

10 January 2020 – started writing the reference_awgplot section for the Programming References chapter.

5 January 2020 – added bullet-list of relevant programming principles to the Programming References section.

4 January 2020 – added Programming References chapter, with section on plotting simple sine waves to the console.

23 May 2019 – added some clarifying comments to the Full Tutorial, and added more questions.

22 May 2019 – added questions.

21 May 2019 – added Derivation on Euler's Relation, showing how $e^{jx} = \cos x + j \sin x$.

August 2018 – added content to the Introduction.

June 2018 – edited oscilloscope image to show triggering slope.

May 2018 – added a Simplified Tutorial chapter.

September 2017 – document first created.

Index

ω, 19
π, 19

AC, 18
Adding quantities to a qualitative problem, 112
Alternating current, 18
Angular velocity, 19
Annotating diagrams, 111
Arctangent, 15

C++, 50
Checking for exceptions, 112
Checking your work, 112
Code, computer, 121
Compiler, C++, 50
Complex arithmetic, "by hand", 45
Complex number, 10, 28, 45
Computer programming, 49
Cosine function, 18
Cycle, 18

Dimensional analysis, 111

Edwards, Tim, 122
Euler's Constant, 41
Euler's Relation, 11, 29, 40
Euler, Leonhard, 11, 29, 40
Excel, Microsoft, 72

Factorial, 40
FORTRAN, programming language, 68
Fourier transform, 20
Frequency, 18

Generator, 9
gnuplot, 73, 74
Graph values to solve a problem, 112
Greenleaf, Cynthia, 85

How to teach with these modules, 119
Hwang, Andrew D., 123

Identify given data, 111
Identify relevant principles, 111
Intermediate results, 111
Interpreter, Python, 54
Inverted instruction, 119

Java, 51

Kirchhoff's Voltage Law, 21
Knuth, Donald, 122

Lamport, Leslie, 122
Limiting cases, 112

Metacognition, 90
Microsoft Excel, 72
Moolenaar, Bram, 121
Murphy, Lynn, 85

Number, complex, 28

Open-source, 121

Pascal, programming language, 68
Phasor, 13, 25
Phasor diagram, 33
Polar form, 10, 33, 45
Problem-solving: annotate diagrams, 111
Problem-solving: check for exceptions, 112
Problem-solving: checking work, 112
Problem-solving: dimensional analysis, 111
Problem-solving: graph values, 112
Problem-solving: identify given data, 111
Problem-solving: identify relevant principles, 111
Problem-solving: interpret intermediate results, 111

Problem-solving: limiting cases, 112
Problem-solving: qualitative to quantitative, 112
Problem-solving: quantitative to qualitative, 112
Problem-solving: reductio ad absurdum, 112
Problem-solving: simplify the system, 62, 111
Problem-solving: thought experiment, 111
Problem-solving: track units of measurement, 111
Problem-solving: visually represent the system, 111
Problem-solving: work in reverse, 112
Programming, computer, 49
Pythagorean theorem, 15
Python, 54

Qualitatively approaching a quantitative problem, 112

Radian, 19
Radians per second, 19
Reading Apprenticeship, 85
Rectangular form, 10, 33, 45
Reductio ad absurdum, 112, 118, 119
RMS, 18
Root-Mean-Square, 18

Scalar number, 10
Schoenbach, Ruth, 85
Scientific method, 90
Simplifying a system, 62, 111
Sine function, 18
Sine wave, 18
Sinusoidal, 18, 20
Socrates, 118
Socratic dialogue, 119
Source code, 50
SPICE, 85
Spreadsheet, 72
Stallman, Richard, 121
Subroutine, 68

Thought experiment, 111
Torvalds, Linus, 121

Unit phasor, 29
Units of measurement, 111

Visualizing a system, 111

Whitespace, C++, 50, 51
Whitespace, Python, 57
Winding, 9
Work in reverse to solve a problem, 112
WYSIWYG, 121, 122