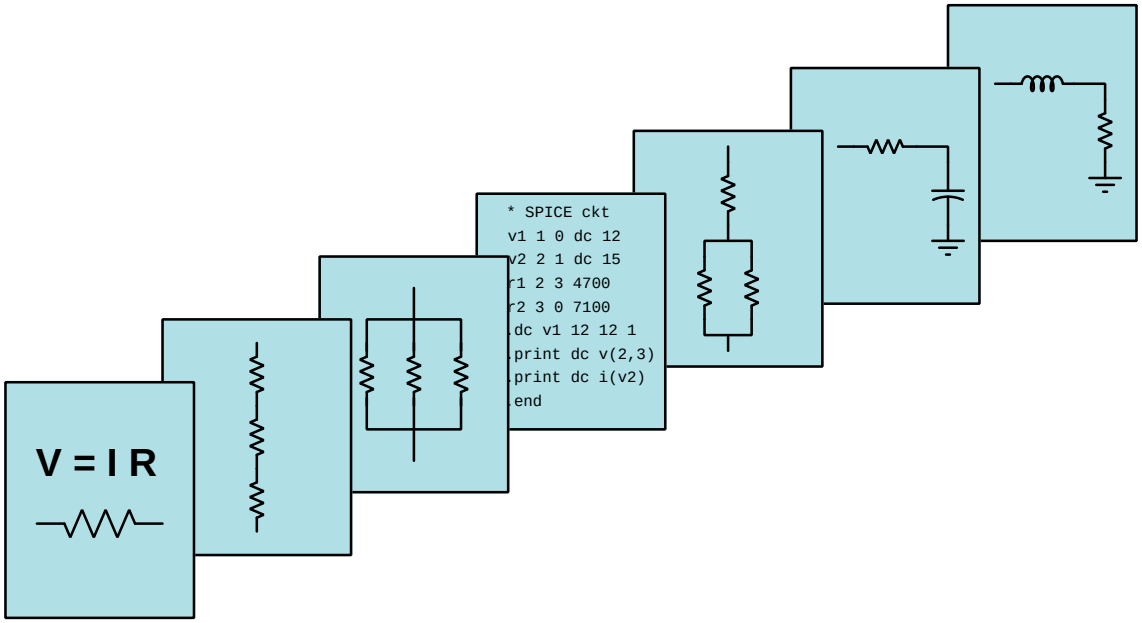
Modular Electronics Learning (ModEL) project



Digital Signal Processing

© 2025-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 17 March 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**

2 Case Tutorial **5**
2.1 Example: . . . 6
2.2 Example: . . . 7
2.3 Example: . . . 8
2.4 Example: . . . 9
2.5 Example: . . . 10

3 Tutorial **11**
3.1 Analog-digital signal conversion . . . 11
3.2 Data types . . . 11
3.3 Data buffering . . . 11
3.4 Digitized signal frequency . . . 11
3.5 Fourier transform functions . . . 11
3.6 Complex numbers . . . 11
3.7 Digital signal mixing . . . 11
3.8 Decimation . . . 11
3.9 Interpolation . . . 11
3.10 Digital signal filtering . . . 11
3.11 Window functions . . . 11

4 Derivations and Technical References **13**
4.1 Definition of negative frequency . . . 14

5 Programming References **19**
5.1 Programming in C++ . . . 20
5.2 Programming in Python . . . 24
5.3 Discrete Fourier Transform algorithm in C++ . . . 29
5.3.1 DFT of a square wave . . . 32
5.3.2 DFT of a sine wave . . . 33
5.3.3 DFT of a delta function . . . 34
5.3.4 DFT of two sine waves . . . 36
5.3.5 DFT of an amplitude-modulated sine wave . . . 37

5.3.6 DFT of a full-rectified sine wave . . . 38
5.4 Spectrum analyzer in C++ . . . 39
5.4.1 Spectrum of a square wave . . . 41
5.4.2 Spectrum of a sine wave . . . 42
5.4.3 Spectrum of a sine wave product . . . 43
5.4.4 Spectrum of an impulse . . . 44
5.5 Crude low-pass filter modeled in C++ . . . 45
5.6 Crude high- and low-pass filters modeled in C++ . . . 46

6 Questions **47**
6.1 Conceptual reasoning . . . 51
6.1.1 Reading outline and reflections . . . 52
6.1.2 Foundational concepts . . . 53
6.1.3 First conceptual question . . . 54
6.1.4 Second conceptual question . . . 54
6.1.5 Applying foundational concepts to ??? . . . 55
6.1.6 Explaining the meaning of calculations . . . 56
6.1.7 Explaining the meaning of code . . . 57
6.2 Quantitative reasoning . . . 58
6.2.1 Miscellaneous physical constants . . . 59
6.2.2 Introduction to spreadsheets . . . 60
6.2.3 First quantitative problem . . . 63
6.2.4 Second quantitative problem . . . 63
6.2.5 ??? simulation program . . . 63
6.3 Diagnostic reasoning . . . 64
6.3.1 First diagnostic scenario . . . 64
6.3.2 Second diagnostic scenario . . . 65

A Problem-Solving Strategies **67**

B Instructional philosophy **69**
B.1 First principles of learning . . . 70
B.2 Proven strategies for instructors . . . 71
B.3 Proven strategies for students . . . 73
B.4 Design of these learning modules . . . 74

C Tools used **77**

D Creative Commons License **81**

E References **89**

F Version history **91**

Index **91**

Chapter 1

Introduction

Chapter 2

Case Tutorial

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

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

2.1 Example:

2.2 Example:

2.3 Example:

2.4 Example:

2.5 Example:

Chapter 3

Tutorial

3.1 Analog-digital signal conversion

3.2 Data types

3.3 Data buffering

3.4 Digitized signal frequency

3.5 Fourier transform functions

3.6 Complex numbers

3.7 Digital signal mixing

3.8 Decimation

3.9 Interpolation

3.10 Digital signal filtering

3.11 Window functions

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 Definition of negative frequency

A bewildering concept one encounters when determining transfer functions for electrical circuits is something known as *negative frequency*. We see this in the transfer functions of circuits containing both inductance and capacitance, where multiple poles exist having imaginary (or complex) values. In such cases, the poles having imaginary values always exist as pairs with equal positive and negative imaginary frequency values.

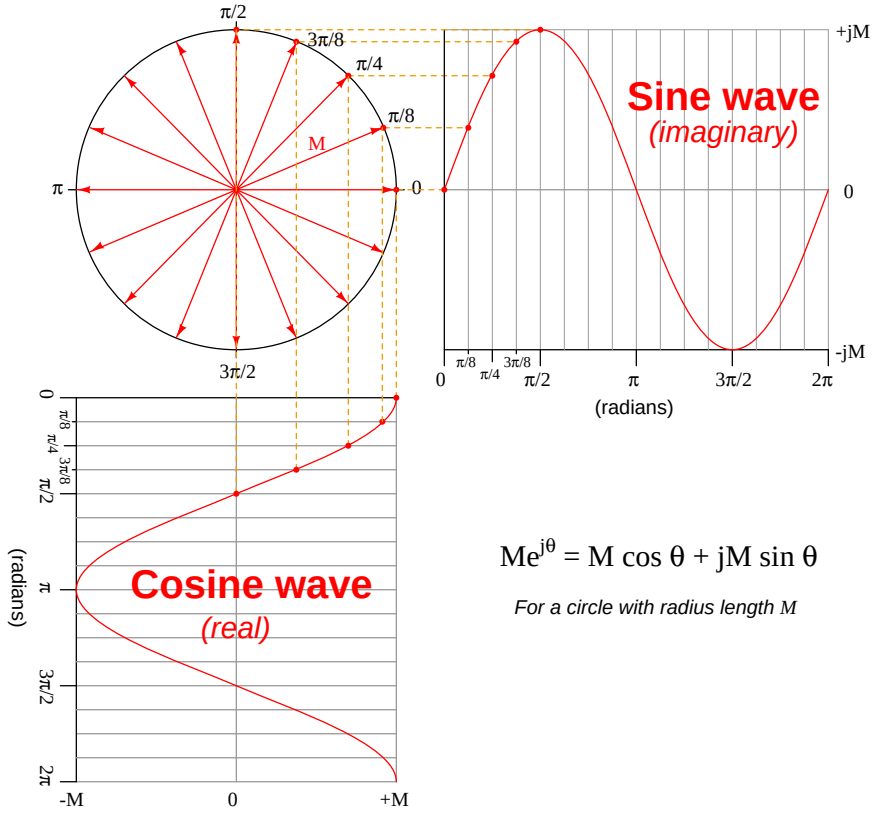
Complex frequency, as we know, consists of both real and imaginary terms, each measured in units of "per second" or s^{-1}. The real portion of a complex frequency value describes how rapidly the signal grows or decays over time, while the imaginary portion describes how rapidly the signal oscillates over time. While the distinction between a positive and negative *real* value for a complex frequency is easy to grasp – positive being growth over time and negative being decay over time – the distinction between positive versus negative imaginary (oscillatory) complex-frequency values is less clear.

Positive versus negative oscillatory frequency is a direct consequence of defining sinusoidal oscillations by Euler's Relation, which shows the mathematical equivalence between imaginary-exponential functions and trigonometric (sine and cosine) functions:

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

In this basic form of Euler's Relation, the angle (θ) is necessarily expressed in mathematically natural unit of *radians* as opposed to the more familiar unit of *degrees* most people use. To see how Euler's Relation gives rise to the possibility of both positive and negative frequency, it helps to graph the results of Euler's Relation as we will do next.

When we mathematically define sinusoidal oscillations by $e^{j\theta}$ we necessarily invoke *two* sinusoidal functions, cosine and sine. These two trigonometric functions, when associated with the real and imaginary axes of a graph, result in a *circle* being traced by the real (horizontal, cosine) and imaginary (vertical, sine) function values as shown in the following diagram:



At an angle (θ) of zero the value of cosine is +1 and the value of sine is zero, placing the coordinate point at the right-hand edge of the circle (marked by a 0). As the angle grows in a positive direction, the cosine function begins to decrease while the sine function increases. The resulting progression of that coordinate point along the circle's circumference is in the counter-clockwise direction, as indicated by the progressively larger angle values shown in the diagram ($\frac{\pi}{8}$, $\frac{\pi}{4}$, etc.).

The fact that combining two out-of-phase sinusoidal functions results in a definite *rotation* should be familiar to anyone who has ever studied polyphase AC circuits. This is precisely why polyphase AC circuits were originally invented, and that was to make possible the creation of a *rotating magnetic field* by sets of wire coils energized by the out-of-phase AC currents. When two or more out-of-phase sinusoids interact within a two-dimensional space, the result is something that *moves* circularly. Any single sinusoid would be incapable of expressing a definite rotational direction, but merely a back-and-forth oscillation.

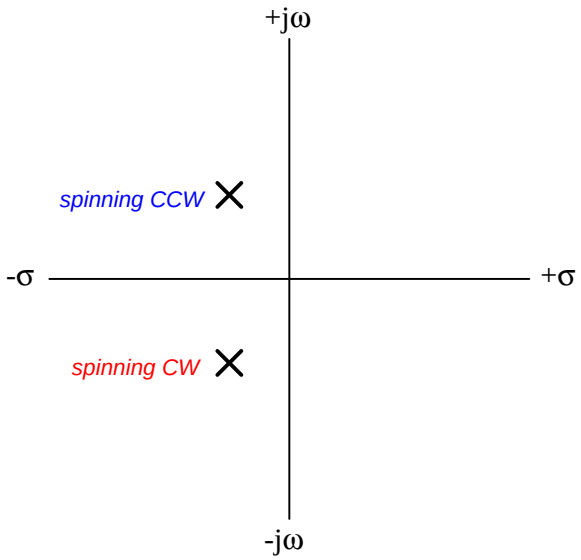
Euler's Relation, then, with its two out-of-phase (by 90 degrees, or $\frac{\pi}{2}$ radians) sinusoids traces a circle with a counter-clockwise rotation as the angle θ grows larger and more positive. A *negative* frequency, then, would have to be one where the direction of rotation is opposite: clockwise instead of counter-clockwise. This is precisely what is being expressed by a transfer function yielding pairs of complex frequency values where the imaginary portions are opposite in sign: *equal and opposite positive- versus negative-rotation oscillations.*

We may mathematically express these positive- and negative-rotating signals by modifying Euler's Relation:

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

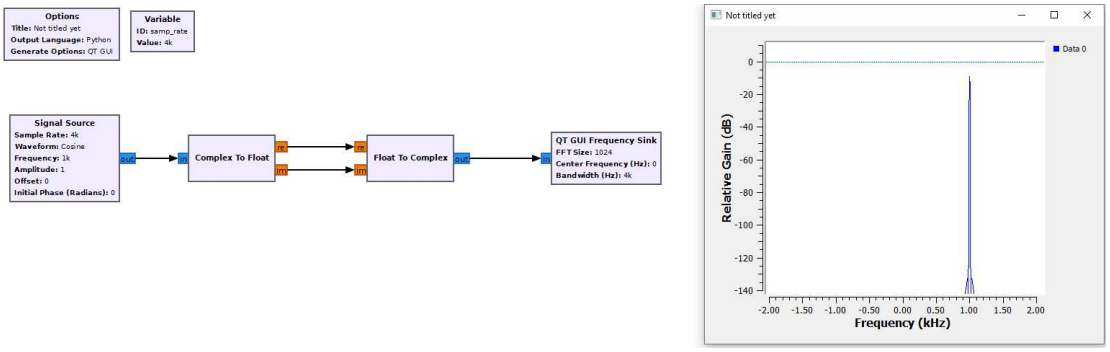
The $\pm$ symbol suggests both positive and negative imaginary terms, which is what we get when solving for roots of a transfer function with a negative discriminant quantity. A negative discriminant in the quadratic equation means the square-root will be of a negative quantity (i.e. imaginary), and then the $\pm$ sign in the quadratic equation results in positive *and* negative imaginary portions for the resulting complex frequency.

In the context of transfer functions, it means the system will have the exact same response to a signal having a negative oscillatory frequency as it would to a signal having a positive oscillatory frequency, which is why these exist in $\pm$ pairs and not alone. For example, if a rotating machine's vibration peaks at a particular shaft speed (i.e. a pole for the vibration transfer function) while spinning clockwise, its vibration will also peak when the shaft spins the same speed in the opposite direction. Those two poles lying opposite from each other across the real axis of the s plane describe the machine's ability to resonate equally well in both rotational directions:

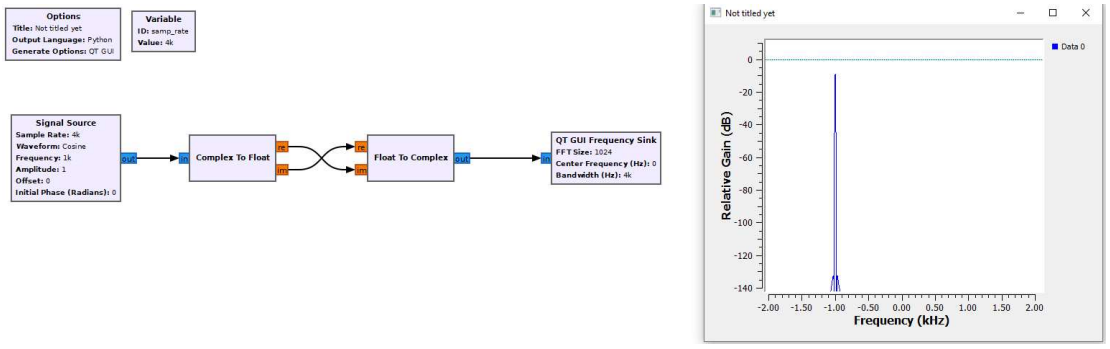


Similarly, an LC "tank" network will resonate if excited by the output of an AC generator at just the right frequency, regardless of which way the generator spins at that speed.

In digital signal processing where sinusoidal signals are often represented as complex numbers, we may demonstrate positive and negative frequencies by taking a simulated signal source and separating its complex output into two floating-point outputs, then recombining those floating-point outputs into complex again and plotting the result on a spectrum display. When the real and imaginary floating-point number streams are recombined in the same manner as they were separated, the resulting complex frequency will have the same (positive) value of +1 kHz as the original source value as shown[1] below:



However, if we swap real and imaginary components of the original complex signal value when recombining into a complex number, the result is a negative frequency of −1 kHz:



Changing the imaginary sign of a complex frequency is a very useful technique in digital signal processing, and as we can see here it may be accomplished by the same technique we would use to reverse the rotation of a polyphase electric motor: in the case of a two-phase motor we would simply swap connections between the two sets of stator coils (windings), or in a three-phase motor swap any two out of the three line conductors.

[1] The digital signal processing software used to create these signal "flowgraphs" is `GNU Radio Companion`.

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 Discrete Fourier Transform algorithm in C++

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

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

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

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

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

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

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

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

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

float f(int x);

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

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

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

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

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

  return 0;
}

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

  else
    return -1.0;
}
```

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

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

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

5.3.1 DFT of a square wave

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

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

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

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

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

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

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

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

5.3.2 DFT of a sine wave

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

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

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

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

5.3.3 DFT of a delta function

As another test of our DFT algorithm, we will re-code `f(x)` to output a *delta function*, which is nothing more than the briefest of impulses. A delta function consists of a "spike"[7] at time zero followed (and preceded) by values of zero:

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

  else
    return 0.0;
}
```

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

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

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

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

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

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

5.3.4 DFT of two sine waves

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

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

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

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

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

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

5.3.5 DFT of an amplitude-modulated sine wave

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

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

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

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

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

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

5.3.6 DFT of a full-rectified sine wave

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

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

The result is shown here:

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

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

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

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

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

5.4 Spectrum analyzer in C++

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

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

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

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

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

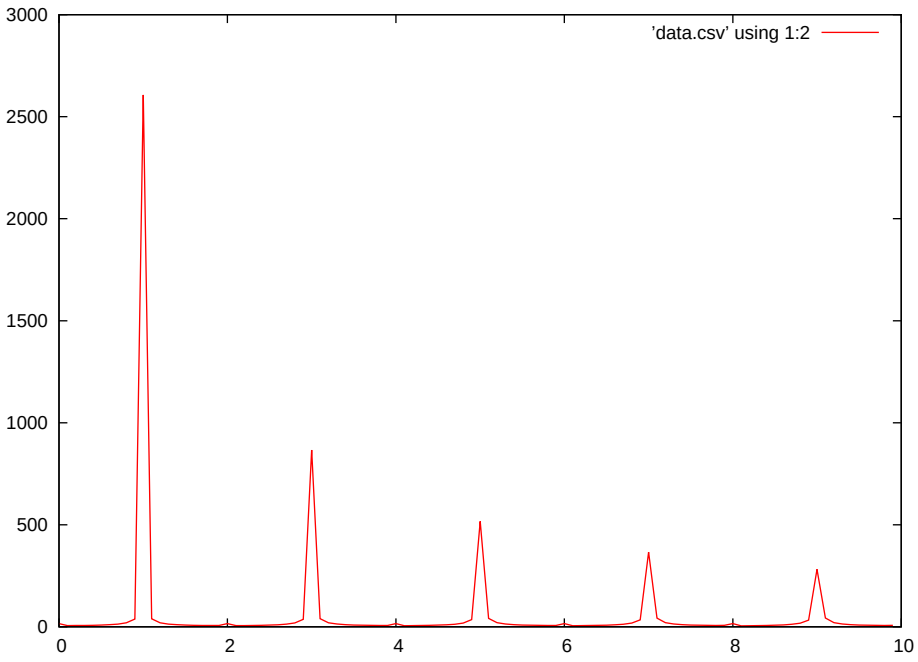
Then, after the comma-separated value file was populated with data from the C++ program's execution, I run `gnuplot` using the following command:

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

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

#define MAX 4096
#define CYCLES 10

float f(int x);

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

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

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

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

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

  return 0;
}

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

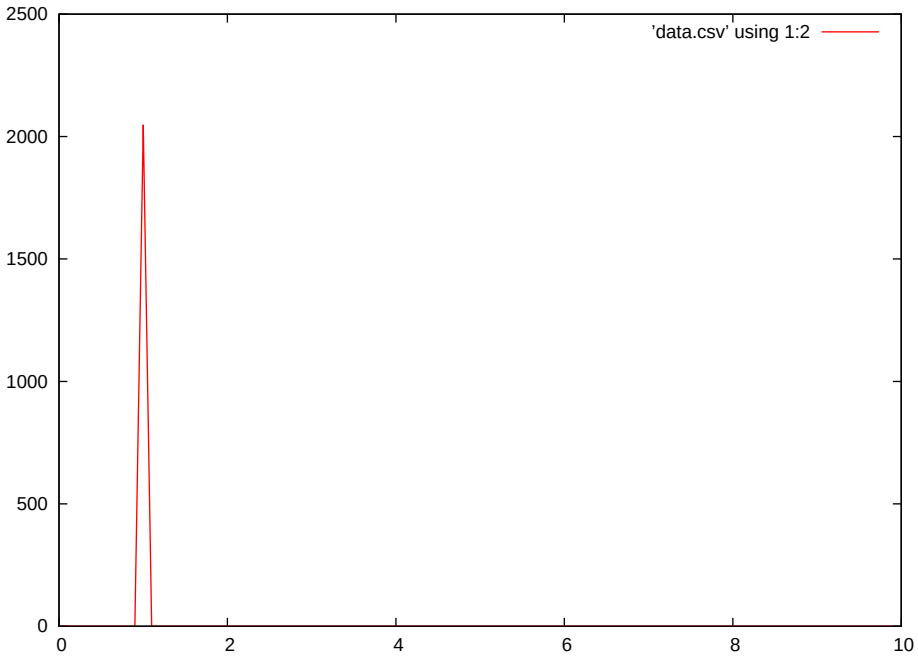
5.4.1 Spectrum of a square wave

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

  else
    return -1.0;
}
```



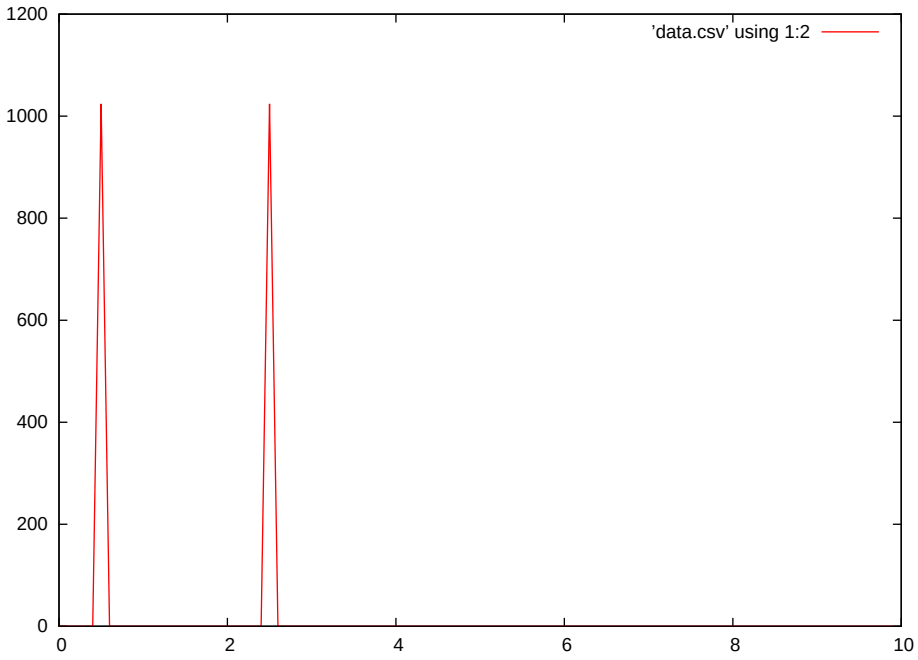
5.4.2 Spectrum of a sine wave

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



5.4.3 Spectrum of a sine wave product

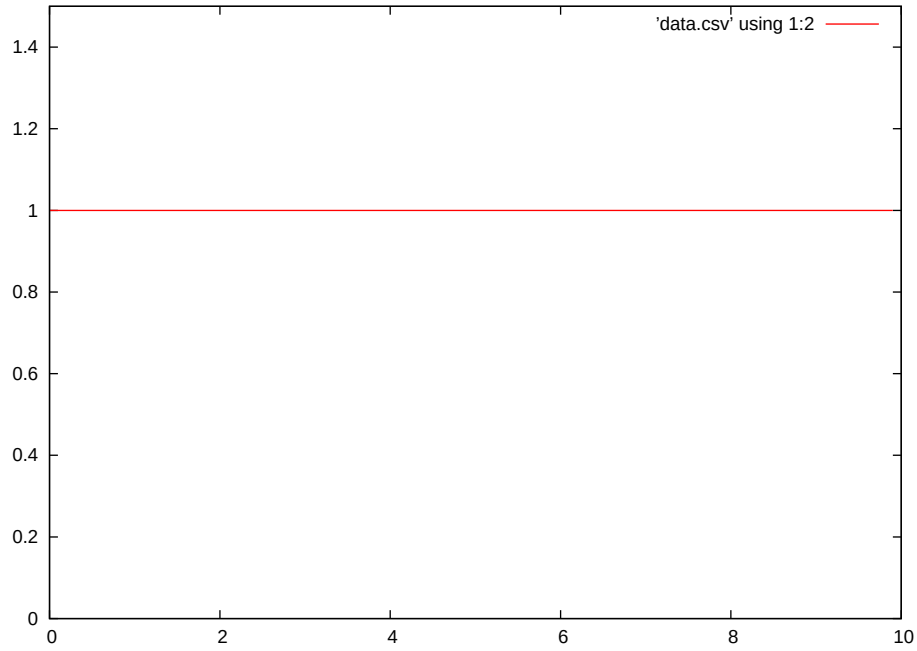
```
float f(int x) // Product of f and 1.5f sine waves
{
  return sin(CYCLES*x*2*M_PI/MAX) * sin(CYCLES*1.5*x*2*M_PI/MAX);
}
```



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

5.4.4 Spectrum of an impulse

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

  else
    return 0;
}
```



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

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

5.5 Crude low-pass filter modeled in C++

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

int main(void)
{
  float wave1[30] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                     0.0, 0.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, 10.0, 10.0,
                     10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0};

  float wave2[30] = {0.0, 10.0, 0.0, 10.0, 0.0, 10.0, 0.0, 10.0, 0.0, 10.0,
                     0.0, 10.0, 0.0, 10.0, 0.0, 10.0, 0.0, 10.0, 0.0, 10.0,
                     0.0, 10.0, 0.0, 10.0, 0.0, 10.0, 0.0, 10.0, 0.0, 10.0};
  float *x = wave1;
  float aout = 0.0;
  int n;

  for (n = 0 ; n < 30 ; ++n)
  {
    aout = (x[n] - aout)/3 + aout;
    cout << n << " , " << x[n] << " , " << aout << endl;
  }

  return 0;
}
```

5.6 Crude high- and low-pass filters modeled in C++

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

#define STEPS 500

int main (void)
{
  float in[STEPS], lp = 0.0, hp;
  float peak = 10.0, freq = 15.0, sigma;
  int n;

  sigma = 2.0 / STEPS;  // Rate in units of Hz or s^-1

  for (n = 0 ; n < STEPS ; ++n)
    in[n] = peak * sin(freq * n / STEPS);  // Sine wave input

  for (n = 0 ; n < STEPS ; ++n)  // Square wave input
  {
    if (n < 10)
      in[n] = 0.0;

    else
      in[n] = 10.0;
  }

  for (n = 0 ; n < STEPS ; ++n)
  {
    hp = in[n] - lp;
    lp = ((in[n] - lp) * sigma) + lp;
    cout << n << " , " << in[n] << " , " << lp << " , " << hp << endl;
  }

  return 0;
}
```

Chapter 6

Questions

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

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

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

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

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

General challenges following tutorial reading

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

General follow-up challenges for assigned problems

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

General follow-up challenges for experiments or projects

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

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

6.1 Conceptual reasoning

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

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

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

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

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

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

6.1.1 Reading outline and reflections

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

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

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

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

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

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

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

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

6.1.2 Foundational concepts

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

Energy

Conservation of Energy

Simplification as a problem-solving strategy

Thought experiments as a problem-solving strategy

Limiting cases as a problem-solving strategy

Annotating diagrams as a problem-solving strategy

Interpreting intermediate results as a problem-solving strategy

Graphing as a problem-solving strategy

Converting a qualitative problem into a quantitative problem

Converting a quantitative problem into a qualitative problem

Working "backwards" to validate calculated results

Reductio ad absurdum

Re-drawing schematics as a problem-solving strategy

Cut-and-try problem-solving strategy

Algebraic substitution

???

6.1.3 First conceptual question

Challenges

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

6.1.4 Second conceptual question

Challenges

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

6.1.5 Applying foundational concepts to ???

Identify which foundational concept(s) apply to each of the declarations shown below regarding the following circuit. If a declaration is true, then identify it as such and note which concept supports that declaration; if a declaration is false, then identify it as such and note which concept is violated by that declaration:

(Under development)

- ???
- ???
- ???
- ???

Here is a list of foundational concepts for your reference: **Conservation of Energy**, **Conservation of Electric Charge**, **behavior of sources vs. loads**, **Ohm's Law**, **Joule's Law**, **effects of open faults**, **effect of shorted faults**, **properties of series networks**, **properties of parallel networks**, **Kirchhoff's Voltage Law**, **Kirchhoff's Current Law**. More than one of these concepts may apply to a declaration, and some concepts may not apply to any listed declaration at all. Also, feel free to include foundational concepts not listed here.

Challenges

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

6.1.6 Explaining the meaning of calculations

Below is a quantitative problem where all the calculations have been performed for you, but all variable labels, units, and other identifying data are unrevealed. *Assign proper meaning* to each of the numerical values, identify the correct unit of measurement for each value as well as any appropriate metric prefix(es), explain the significance of each value by describing where it "fits" into the circuit being analyzed, and identify the general principle employed at each step:

Schematic diagram of the ??? circuit:

(Under development)

Calculations performed in order from first to last:

1. $x + y = z$
2. $x + y = z$
3. $x + y = z$
4. $x + y = z$
5. $x + y = z$
6. $x + y = z$

Challenges

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

6.1.7 Explaining the meaning of code

Shown below is a schematic diagram for a ??? circuit, and after that a source-code listing of a computer program written in the ??? language simulating that circuit. Explain the purpose of each line of code relating to the circuit being simulated, identify the correct unit of measurement for each computed value, and identify all foundational concepts of electric circuits (e.g. Ohm's Law, Kirchhoff's Laws, etc.) employed in the program:

Schematic diagram of the ??? circuit:

(Under development)

Code listing:

```
#include <stdio.h>

int main (void)
{

  return 0;
}
```

Challenges

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

6.2 Quantitative reasoning

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

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

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

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

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

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

6.2.1 Miscellaneous physical constants

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.2 Introduction to spreadsheets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.3 First quantitative problem

Challenges

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

6.2.4 Second quantitative problem

Challenges

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

6.2.5 ??? simulation program

Write a text-based computer program (e.g. C, C++, Python) to calculate ???

Challenges

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

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 First diagnostic scenario

Challenges

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

6.3.2 Second diagnostic scenario

Challenges

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

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

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.

17 March 2026 – added a Derivation section on negative frequency.

28 July 2025 – added some C++ programming examples showing how low-pass and high-pass filters may be modeled.

22 April 2025 – document first created, with negligible content.

Index

Adding quantities to a qualitative problem, 68
Annotating diagrams, 67

C++, 20
Checking for exceptions, 68
Checking your work, 68
Code, computer, 77
Compiler, C++, 20
Computer programming, 19

Delta impulse function, 34
DFT, 29
Dimensional analysis, 67
Dirac delta function, 34
Discrete Fourier Transform, 29
Discriminant, 16

Edwards, Tim, 78
Euler's Relation, 14

GNU Radio Companion software, 17
Graph values to solve a problem, 68
Greenleaf, Cynthia, 47

How to teach with these modules, 75
Hwang, Andrew D., 79

Identify given data, 67
Identify relevant principles, 67
Impulse function, 34
Intermediate results, 67
Interpreter, Python, 24
Inverted instruction, 75

Java, 21

Knuth, Donald, 78

Lamport, Leslie, 78

Limiting cases, 68

Metacognition, 52
Moolenaar, Bram, 77
Murphy, Lynn, 47

Open-source, 77

Problem-solving: annotate diagrams, 67
Problem-solving: check for exceptions, 68
Problem-solving: checking work, 68
Problem-solving: dimensional analysis, 67
Problem-solving: graph values, 68
Problem-solving: identify given data, 67
Problem-solving: identify relevant principles, 67
Problem-solving: interpret intermediate results, 67
Problem-solving: limiting cases, 68
Problem-solving: qualitative to quantitative, 68
Problem-solving: quantitative to qualitative, 68
Problem-solving: reductio ad absurdum, 68
Problem-solving: simplify the system, 67
Problem-solving: thought experiment, 67
Problem-solving: track units of measurement, 67
Problem-solving: visually represent the system, 67
Problem-solving: work in reverse, 68
Programming, computer, 19
Pythagorean theorem, 29
Python, 24

Qualitatively approaching a quantitative problem, 68

Reading Apprenticeship, 47
Reductio ad absurdum, 68, 74, 75

Schoenbach, Ruth, 47

Scientific method, 52
Simplifying a system, 67
Socrates, 74
Socratic dialogue, 75
Source code, 20
SPICE, 47
Stallman, Richard, 77

Thought experiment, 67
Torvalds, Linus, 77

Units of measurement, 67

Visualizing a system, 67

Whitespace, C++, 20, 21
Whitespace, Python, 27
Work in reverse to solve a problem, 68
WYSIWYG, 77, 78