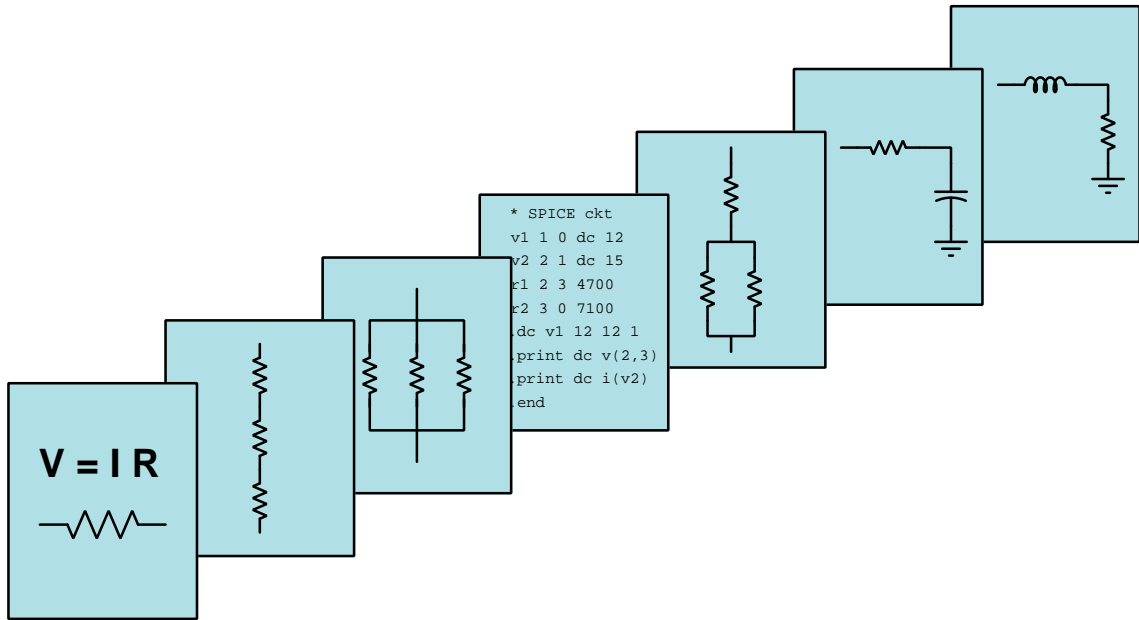


MODULAR ELECTRONICS LEARNING (MODEL) PROJECT



SINGLE-STAGE FET AMPLIFIERS

© 2019-2020 BY TONY R. KUPHALDT – UNDER THE TERMS AND CONDITIONS OF THE
CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL PUBLIC LICENSE

LAST UPDATE = 9 JANUARY 2020

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	Tutorial	5
3	Historical References	7
3.1	???	8
4	Derivations and Technical References	9
4.1	Derivation of ???	10
4.2	??? Reference	11
5	Programming References	13
5.1	Programming in C++	14
5.2	Programming in Python	18
5.3	Modeling a common-source amplifier using C++	22
6	Questions	31
6.1	Conceptual reasoning	35
6.1.1	Reading outline and reflections	36
6.1.2	Foundational concepts	37
6.1.3	First conceptual question	37
6.1.4	Second conceptual question	37
6.2	Quantitative reasoning	38
6.2.1	Introduction to spreadsheets	39
6.2.2	First quantitative problem	42
6.2.3	Second quantitative problem	42
6.3	Diagnostic reasoning	43
6.3.1	First diagnostic scenario	43
6.3.2	Second diagnostic scenario	44
7	Projects and Experiments	45
7.1	Recommended practices	45
7.1.1	Safety first!	46
7.1.2	Other helpful tips	48
7.1.3	Terminal blocks for circuit construction	49

<i>CONTENTS</i>	1
7.1.4 Conducting experiments	52
7.1.5 Constructing projects	56
7.2 Experiment: (first experiment)	57
7.3 Project: (first project)	58
A Problem-Solving Strategies	59
B Instructional philosophy	61
C Tools used	67
D Creative Commons License	71
E References	79
F Version history	81
Index	81

Chapter 1

Introduction

(Introductory text goes here)

Difficult concepts for students include the following:

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

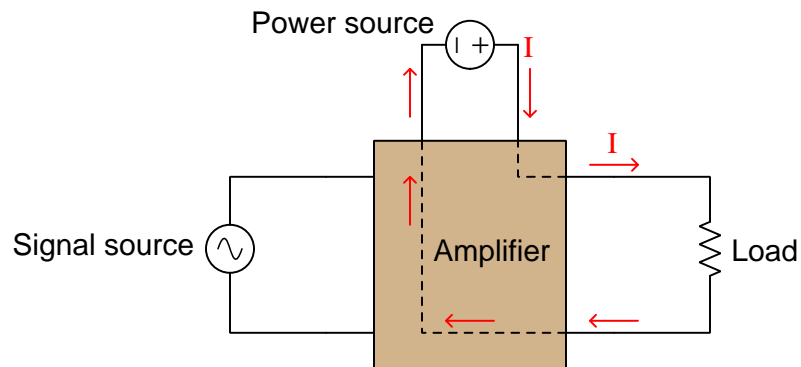
Chapter 2

Tutorial

An *amplifier* is any device allowing a small amount of energy to exert control over a larger amount of energy. Every electronic amplifier, regardless of the specific technology used, may be thought of as having three different sets of connection points:

1. Signal input (connects to signal source)
2. Signal output (connects to load)
3. Power (connects to electrical power source)

A simplified diagram of a generic electronic amplifier appears below, without any reference to the active components contained within:



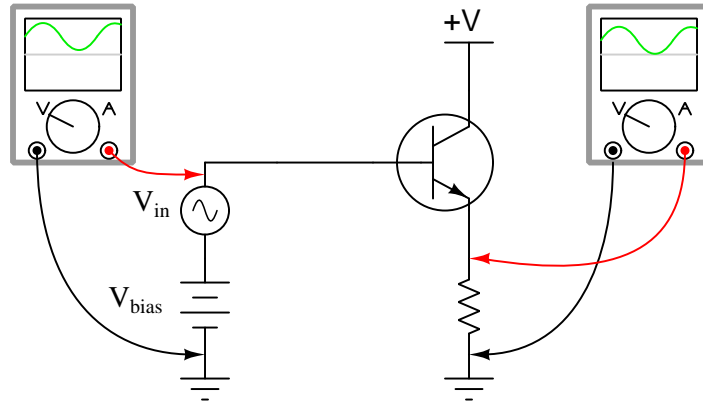
Electronic components such as transistors (or, in previous eras, vacuum tubes) comprise the heart of amplifiers, giving the input signal control over the flow of energy from power source to load.

One important parameter for any amplifier is *gain*, defined simply as the ratio of controlled versus controlling quantities, usually symbolized by the variable A . The value of A tells us how many times larger an amplifier's output signal is compared to its input signal. This ratio of output to input may be expressed in multiple ways. For example, we may speak of an amplifier having a *power* gain, a *voltage* gain, and a *current* gain, each gain calculated as the respective ratio of output to input for the specified quantity:

$$A_P = \frac{P_{out}}{P_{in}} \quad A_V = \frac{V_{out}}{V_{in}} \quad A_I = \frac{I_{out}}{I_{in}}$$

Gain is always unitless because it is a ratio between quantities having identical units of measurement (e.g. Watts out versus Watts in, Amperes out versus Amperes in).

Amplifiers are often used to boost the amplitude of *AC* signals, which poses a technical problem because most active components (e.g. transistors, vacuum tubes) are fundamentally DC-only devices. A common solution to this problem is to *bias* the AC signal input to an amplifier with a DC "offset" until it becomes "shifted" so far that its polarity never reverses. This technique is shown in the following schematic diagram, where the AC signal (V_{in}) is connected in series with a DC voltage source (V_{bias}) sufficient to offset the signal waveform entirely above the "zero" line on the oscilloscope display, allowing the transistor to operate on a non-alternating voltage signal:



This amplifier's output signal faithfully represents the wave-shape of the AC input signal, and is also "biased" above zero as a result of the transistor being incapable of controlling a bidirectional current.

When the amplifier circuit in question amplifies an AC signal in this manner – where the input and output signals are both biased, and not necessarily biased by the same amount of DC offset – it becomes useful to define gain in terms of *variations* of power, voltage, or current. This is referred to as the *AC gain* of the circuit, expressed as ratios of *changes* denoted by the Greek capital letter delta (Δ). For example, an AC gain of 2 means the output rises and falls twice as much as the input rises and falls:

$$A_P = \frac{\Delta P_{out}}{\Delta P_{in}} \quad A_V = \frac{\Delta V_{out}}{\Delta V_{in}} \quad A_I = \frac{\Delta I_{out}}{\Delta I_{in}}$$

Chapter 3

Historical References

This chapter is where you will find passages and/or illustrations from historical texts related to the module's topic.

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

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

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

3.1 ???

???

???

Chapter 4

Derivations and Technical References

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

4.1 Derivation of ???

4.2 ??? Reference

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 pleasing¹ to view.

¹Although 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 system², 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 Studio³, 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

²A 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*!

³Using 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()
```

⁴Using 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
```

Note how Python defaults to rectangular form for complex quantities. Here we defined a 400 Ohm resistance as a complex value in rectangular form ($400 + j0 \Omega$), then computed capacitive and inductive reactances at 60 Hz and defined each of those as complex (phasor) values ($0 - jX_c \Omega$ and $0 + jX_l \Omega$, respectively). After that we computed total impedance in series, then total impedance in parallel. Polar-form representation was then shown for the series impedance ($441.717 \Omega \angle -25.102^\circ$). 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 assignment of variables as well as a convenient text record⁶ of all calculations performed which may be easily copied and pasted into a text document for archival.

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

⁶Like 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.

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:

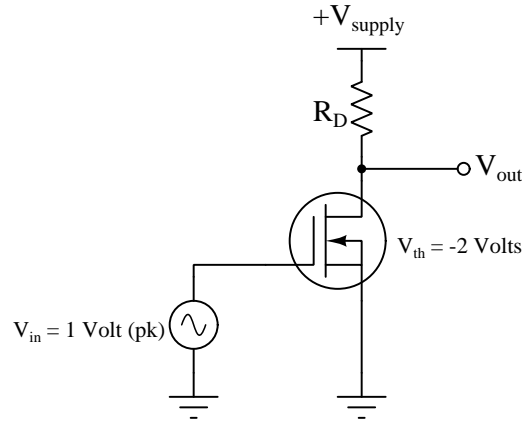
```
Sum
-360.5
Difference
760.5
Product
-112100.0
Quotient
-0.35682426405
```

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.

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 Modeling a common-source amplifier using C++

In this section we will show the time-domain output of a common-source FET amplifier using a simple program written in the C++ language. A schematic diagram of the amplifier is shown here:



The FET used in this example is a depletion-type N-channel MOSFET with a turn-off (i.e. threshold) voltage of -2 Volts. The input voltage source is AC with a peak value of 1 Volt, which means it alternates in sinusoidal fashion reaching peak instantaneous voltage values of $+1$ Volt and -1 Volt. Thus, the input voltage source will never reach a value large enough to fully turn off the MOSFET – it will always remain conductive between drain and source.

We should expect this amplifier to have an *inverting* characteristic, its output voltage falling as input voltage rises. This is because, being an N-channel device, its channel conductivity will enhance as its gate becomes more positive with respect to its source, and that enhanced conductivity draws more current which drops more voltage across resistor R_D and leaves less voltage dropped across the MOSFET's drain and source terminals. In this manner it is analogous to a common-emitter BJT amplifier using a NPN transistor, operating in class-A⁷ mode.

On the next page we will find the C++ code used to simulate the transistor and plot an output waveform to the computer's console.

⁷Class-A refers to a mode of operation for an amplifier where the active component always remains conducting and never cuts off. Class-B mode is where the active component conducts for one-half of the waveform (usually complemented by another active device in the circuit conducting during the other half of the waveform). Class-C mode is where the active component conducts for less than 50% of the waveform's period, typically used in RF circuits with resonant elements able to complete the rest of the waveform's evolution undriven by the active component.

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

int plot(float);

int main (void)
{
    float k, V_in, V_th, V_out, V_supply, R_D, I_D, theta;

    V_supply = 24.0;
    R_D = 12e3;
    V_th = -2.0;
    k = 1.5e-4;

    for (theta = 0 ; theta <= (2 * M_PI) ; theta = theta + 0.2)
    {
        V_in = sin(theta);
        I_D = k * pow(V_in - V_th,2);
        V_out = V_supply - (I_D * R_D);

        if (V_out < 0)
            V_out = 0.0;

        else if (V_out > V_supply)
            V_out = V_supply;

        plot(V_out);
    }

    return 0;
}

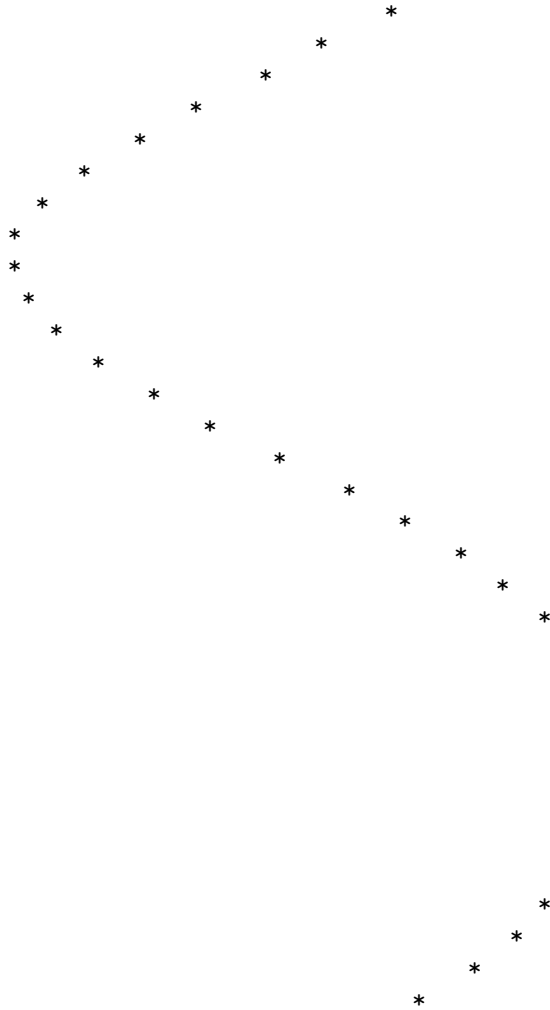
int plot(float x)
{
    int n;

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

    cout << "*" << endl;

    return 0;
}
```

When compiled and run, the program produces the following output:



As you can see, the waveform is not a perfect sinusoid. This is due to the nonlinear nature of the FET, being a *square-law* device.

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

- Order of execution
- Preprocessor directives, namespaces
- The `main` function: return values, arguments
- Delimiter characters (e.g. `{ } ;`)
- Variable types (`float`, `int`, `bool`), names, and declarations
- Variable assignment/initialization (`=`)
- Conditional statements (`if`, `else if`)
- Comparison (`<`, `>`, `<=`)
- Loops (`for`)
- Increment/decrement operators (`++`)
- Arithmetic functions (`sin`)
- Custom functions: prototyping, return values, arguments

As with all C/C++ programs, the general order of execution is left-to-right, top-to-bottom. At the very top we see the *preprocessor directives* (`#include`) informing the compiler software of the function libraries our program will need access to, and the definitions of those functions. The `namespace` statement declares the scope of function names.

Below this we have the line `int plot(float);` which is a *prototyping* statement for our own custom function. Languages such as C and C++ encourage writing modular code, where small sections of code are contained within their own “functions” and those functions may be called arbitrarily by other functions. Every C/C++ program has a `main` function, which marks the initial point of execution when the program is first run. The boundaries for each function are delimited by “brace” symbols (`{ }`). Lines of code within each function are shown in this listing indented, but this indentation is strictly for the sake of human readability – the compiler pays no attention to indentation. Each single line of code is terminated by a semicolon character (`;`) which tells the compiler where one line ends and another begins.

Functions in C and C++ may take in data and produce data. In the case of the `main` function it receives no data (`void`) but produces an integer number value (`int`). In the case of our custom function called `plot`, it receives a single floating-point (`float`) argument and returns an integer number (`int`). In both cases the returned integer has a constant value of zero, which may seem pointless, but returning *some* data upon completion is considered good programming practice. If a function does not strictly need to return any data, returning an integer provides a good way for the rest of the program to verify the function executed successfully. It is commonplace to find functions written in such a way that the returned value communicates useful information about the function's execution (e.g. the function value may represent an *error code*).

Integer (`int`) and floating-point `float` are two common types of numbers representable by a computer. Integer numbers are whole numbers and their negative counterparts. Floating-point numbers are essentially scientific notation in binary format, capable of representing fractional quantities, extremely large quantities, and extremely small quantities. Continuous electrical circuit parameters such as voltage and current are best represented by floating-point.

The `float` line at the top of the `main` function declares the use of several floating-point variables in this program. As you can tell by the names that follow, variable names in C/C++ may consist of single alphabetical characters, multiple alphabetical characters, upper- and lower-case characters, and even some types of symbols such as underscore (`_`). Blank spaces are not allowed within variable names, so when a blank space would be esthetically pleasing we use an unobtrusive character such as the underscore. These declaration statements are necessary because it instructs the compiler how much of the computer's memory space to reserve for each variable.

Below the declaration line follow four lines of code, each one "initializing" a variable with some numerical value. In each case a single "equals" symbol (`=`) acts as an *assignment* operator, instructing the computer to set the value of the variable (on the left) to be equal to the value on the right. This is different from the meaning of an "equals" symbol in mathematics, which simply states two expressions to have the same value as one another. In C/C++, an "equals" symbol means the left-hand term will *have its value set equal to* the right-hand expression.

Below the assignment lines we have an interesting construct known as a `for` loop. Like a function, it encapsulates a set of code lines within a pair of "curly-brace" symbols, but unlike a function the purpose of a `for` loop is to repeatedly execute those lines of code on its own without having to be "called" from somewhere else. The terms within the parentheses declare the conditions for repeating the loop. In this case, the loop's repetition is conditioned entirely on the value of the `theta` variable: first setting its value to zero, increasing its value by 0.2 with each iteration, and checking to see that its value is equal to or less than 2π before repeating.

Within the `for` loop we have eight lines of code. The first three simply calculate parameters within the FET amplifier circuit: generating a sine wave (`V_in = sin(theta)`) as the input voltage signal, calculating drain current based on gate-source voltage (`I_D = k * pow(V_in - V_th, 2)`), and finally calculating output voltage (i.e. voltage dropped across the FET's drain and source terminals) by subtracting the resistor's voltage from the supply voltage (`V_out = V_supply - (I_D * R_D)`).

Following these arithmetical expressions we have two *conditional* statements, an `if` and an `else if`. Neither one is terminated by a semicolon because the lines immediately following each are considered part of the same line of code (remember that whitespace on the page is meaningless to C/C++). The parentheses enclose the condition being tested, and the next statement is what will be executed if that condition is true. So, if the output voltage is less than zero (i.e. negative), it gets its value capped at a low limit of zero; if the output voltage exceeds supply voltage, it gets its value capped at a high limit equal to the supply voltage. These conditional statements are included in the `for` loop in order to place realistic limits on the amplifier's output voltage. Without them, the code could potentially calculate values for V_{out} that would be impossible in any real circuit.

The last line repeated by the `for` loop calls our custom function, `plot`. Here is where the program's execution momentarily breaks out of the `for` loop and jumps down to our code for the `plot` function shown at the very bottom of the listing.

Note how when called within the `for` loop, the `plot` function receives an input value from the variable `V_out`. This is because we want the `plot` function to plot a character to the computer's console representing the instantaneous value of the output voltage waveform. Within the `plot` function, however, the input value (i.e. "argument") is renamed to the variable `x` which as a "local" variable is strictly contained to the `plot` function.

Also local to the `plot` function is an integer variable named `n`. This variable is used to govern the repetitions of another `for` loop, starting with an `n` value of zero, continuing so long as `n` is less than three times⁸ the value of `x` (V_{out}), and incrementing by one (`++n`) with each iteration. Unlike the `for` loop within the `main` program which looped eight lines of code, this one only loops a single line of code. Notice how, just like the `if` and `else if` conditional statements, when a solitary line of code is consequent there is no need to enclose it within curly-brace characters. Braces are necessary when multiple lines of code are to be executed as a set.

The purpose of the `plot` function is to print to the computer's console display a series of spaces proportional in number to the instantaneous value of V_{out} and then terminate each line of printed text with a single star (*) character. These print operations utilize the `cout` instruction, part of the standard `iostream` library in C++. When complete, the function returns a value of zero (just for convention's sake) and the flow of execution jumps back to the `for` loop from where it was originally called.

Now that we've seen how this program works, let's use it to run some virtual experiments on the amplifier circuit. After all, what good is a model if we can't test it?

⁸The reason for this multiplier is to obtain a waveform of significant amplitude on the console, given the standard width limit of 80 characters. Without this multiplier, the printed waveform will appear much smaller in amplitude and the fixed resolution of the text-based character display may not show enough detail of the wave's shape to suit our liking.

As you can see, the waveform is shifted so far that it “clips” against the positive supply rail limit. Our C++ model of the amplifier circuit forces clipping to occur by means of the `else if` statement, designed to mimic what a real common-source FET amplifier circuit would do:

```
else if (V_out > V_supply)
    V_out = V_supply;
```

This, of course, is just one simple way we may use our software model to explore the behavior of an amplifier circuit. Other changes we could make include the following:

- Changing the value of resistor R_D
- Changing the transistor’s transconductance (k)
- Changing the supply voltage V_{supply}

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 read¹ the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture², 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.

¹Technical 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 *Reading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms* by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

²Lecture 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 thinking³. 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, write their own outline and 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 **OUTLINE THE TEXT**, as though you were writing a detailed Table of Contents. Feel free to rearrange the order if it makes more sense that way. Prepare to articulate these points in detail and to answer questions from your classmates and instructor. Outlining is a good self-test of thorough reading because you cannot outline what you have not read or do 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

???

6.1.3 First conceptual question

This is the text of the question!

Challenges

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

6.1.4 Second conceptual question

This is the text of the question!

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 cases⁴” for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely⁵ on an answer key!

⁴In 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.

⁵This 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 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.

⁶Spreadsheets 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.

Common⁷ 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 ensure⁸ 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 roots⁹ 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.

⁷Modern 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.

⁸Spreadsheet 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.

⁹Reviewing 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:

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

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

	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 arbitrary¹⁰ – 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.

¹⁰My 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.*

6.2.2 First quantitative problem

This is a description of the problem!

Challenges

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

6.2.3 Second quantitative problem

This is a description of the problem!

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

This is a description of the scenario!

Challenges

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

6.3.2 Second diagnostic scenario

This is a description of the scenario!

Challenges

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

Chapter 7

Projects and Experiments

The following project and experiment descriptions outline things you can build to help you understand circuits. With any real-world project or experiment there exists the potential for physical harm. *Electricity can be very dangerous in certain circumstances, and you should follow proper safety precautions at all times!*

7.1 Recommended practices

This section outlines some recommended practices for all circuits you design and construct.

7.1.1 Safety first!

Electricity, when passed through the human body, causes uncomfortable sensations and in large enough measures¹ will cause muscles to involuntarily contract. The overriding of your nervous system by the passage of electrical current through your body is particularly dangerous in regard to your heart, which is a vital muscle. Very large amounts of current can produce serious internal burns in addition to all the other effects.

Cardio-pulmonary resuscitation (CPR) is the standard first-aid for any victim of electrical shock. This is a very good skill to acquire if you intend to work with others on dangerous electrical circuits. You should never perform tests or work on such circuits unless someone else is present who is proficient in CPR.

As a general rule, any voltage in excess of 30 Volts poses a definitive electric shock hazard, because beyond this level human skin does not have enough resistance to safely limit current through the body. “Live” work of any kind with circuits over 30 volts should be avoided, and if unavoidable should only be done using electrically insulated tools and other protective equipment (e.g. insulating shoes and gloves). If you are unsure of the hazards, or feel unsafe at any time, stop all work and distance yourself from the circuit!

A policy I strongly recommend for students learning about electricity is to *never come into electrical contact² with an energized conductor, no matter what the circuit’s voltage³ level!* Enforcing this policy may seem ridiculous when the circuit in question is powered by a single battery smaller than the palm of your hand, but it is precisely this instilled habit which will save a person from bodily harm when working with more dangerous circuits. Experience has taught me that students who learn early on to be careless with safe circuits have a tendency to be careless later with dangerous circuits!

In addition to the electrical hazards of shock and burns, the construction of projects and running of experiments often poses other hazards such as working with hand and power tools, potential

¹Professor Charles Dalziel published a research paper in 1961 called “The Deleterious Effects of Electric Shock” detailing the results of electric shock experiments with both human and animal subjects. The threshold of perception for human subjects holding a conductor in their hand was in the range of 1 milliampere of current (less than this for alternating current, and generally less for female subjects than for male). Loss of muscular control was exhibited by half of Dalziel’s subjects at less than 10 milliamperes alternating current. Extreme pain, difficulty breathing, and loss of all muscular control occurred for over 99% of his subjects at direct currents less than 100 milliamperes and alternating currents less than 30 milliamperes. In summary, it doesn’t require much electric current to induce painful and even life-threatening effects in the human body! Your first and best protection against electric shock is maintaining an insulating barrier between your body and the circuit in question, such that current from that circuit will be unable to flow through your body.

²By “electrical contact” I mean either directly touching an energized conductor with any part of your body, or indirectly touching it through a conductive tool. The only physical contact you should ever make with an energized conductor is via an electrically insulated tool, for example a screwdriver with an electrically insulated handle, or an insulated test probe for some instrument.

³Another reason for consistently enforcing this policy, even on low-voltage circuits, is due to the dangers that even some low-voltage circuits harbor. A single 12 Volt automobile battery, for example, can cause a surprising amount of damage if short-circuited simply due to the high current levels (i.e. very low internal resistance) it is capable of, even though the voltage level is too low to cause a shock through the skin. Mechanics wearing metal rings, for example, are at risk from severe burns if their rings happen to short-circuit such a battery! Furthermore, even when working on circuits that are simply too low-power (low voltage and low current) to cause any bodily harm, touching them while energized can pose a threat to the circuit components themselves. In summary, it generally wise (and *always* a good habit to build) to “power down” *any* circuit before making contact between it and your body.

contact with high temperatures, potential chemical exposure, etc. You should never proceed with a project or experiment if you are unaware of proper tool use or lack basic protective measures (e.g. personal protective equipment such as safety glasses) against such hazards.

Some other safety-related practices should be followed as well:

- All power conductors extending outward from the project must be *firmly* strain-relieved (e.g. “cord grips” used on line power cords), so that an accidental tug or drop will not compromise circuit integrity.
- All electrical connections must be sound and appropriately made (e.g. soldered wire joints rather than twisted-and-taped; terminal blocks rather than solderless breadboards for high-current or high-voltage circuits). Use “touch-safe” terminal connections with recessed metal parts to minimize risk of accidental contact.
- Always provide overcurrent protection in any circuit you build. *Always*. This may be in the form of a fuse, a circuit breaker, and/or an electronically current-limited power supply.
- Always ensure circuit conductors are rated for more current than the overcurrent protection limit. *Always*. A fuse does no good if the wire or printed circuit board trace will “blow” before it does!
- Always bond metal enclosures to Earth ground for any line-powered circuit. *Always*. Ensuring an equipotential state between the enclosure and Earth by making the enclosure electrically common with Earth ground ensures no electric shock can occur simply by one’s body bridging between the Earth and the enclosure.
- Avoid building a high-energy circuit when a low-energy circuit will suffice. For example, I always recommend beginning students power their first DC resistor circuits using small batteries rather than with line-powered DC power supplies. The intrinsic energy limitations of a dry-cell battery make accidents highly unlikely.
- Use line power receptacles that are GFCI (Ground Fault Current Interrupting) to help avoid electric shock from making accidental contact with a “hot” line conductor.
- Always wear eye protection when working with tools or live systems having the potential to eject material into the air. Examples of such activities include soldering, drilling, grinding, cutting, wire stripping, working on or near energized circuits, etc.
- Always use a step-stool or stepladder to reach high places. Never stand on something not designed to support a human load.
- When in doubt, *ask an expert*. If anything even seems remotely unsafe to you, do not proceed without consulting a trusted person fully knowledgeable in electrical safety.

7.1.2 Other helpful tips

Experience has shown the following practices to be very helpful, especially when students make their own component selections, to ensure the circuits will be well-behaved:

- Avoid resistor values less than $1\text{ k}\Omega$ or greater than $100\text{ k}\Omega$, unless such values are definitely necessary⁴. Resistances below $1\text{ k}\Omega$ may draw excessive current if directly connected to a voltage source of significant magnitude, and may also complicate the task of accurately measuring current since any ammeter's non-zero resistance inserted in series with a low-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Resistances above $100\text{ k}\Omega$ may complicate the task of measuring voltage since any voltmeter's finite resistance connected in parallel with a high-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Similarly, AC circuit impedance values should be between $1\text{ k}\Omega$ and $100\text{ k}\Omega$, and for all the same reasons.
- Ensure all electrical connections are low-resistance and physically rugged. For this reason, one should avoid *compression splices* (e.g. "butt" connectors), solderless breadboards⁵, and wires that are simply twisted together.
- Build your circuit with **testing** in mind. For example, provide convenient connection points for test equipment (e.g. multimeters, oscilloscopes, signal generators, logic probes).
- Design permanent projects with **maintenance** in mind. The more convenient you make maintenance tasks, the more likely they will get done.
- **Always document and save your work.** Circuits lacking schematic diagrams are more difficult to troubleshoot than documented circuits. Similarly, circuit construction is simpler when a schematic diagram precedes construction. Experimental results are easier to interpret when comprehensively recorded. Consider modern videorecording technology for this purpose where appropriate.
- **Record your steps** when troubleshooting. **Talk to yourself** when solving problems. These simple steps clarify thought and simplify identification of errors.

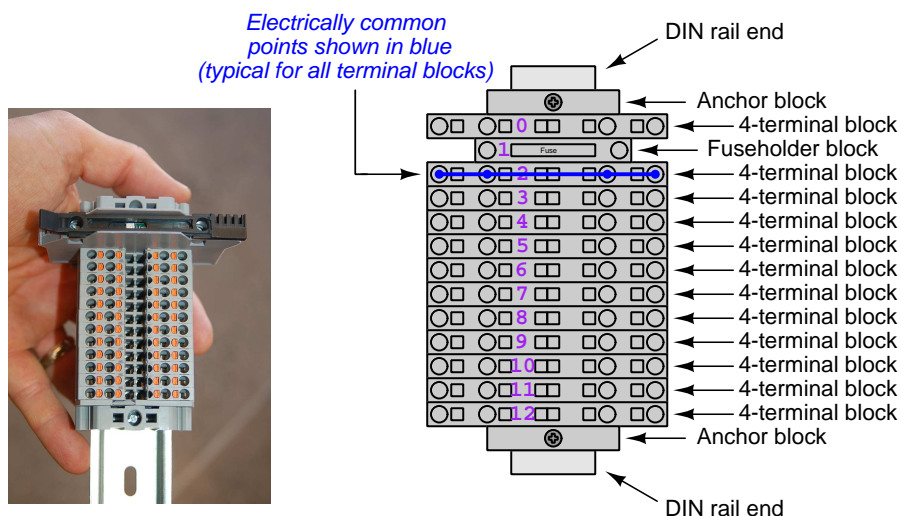
⁴An example of a necessary resistor value much less than $1\text{ k}\Omega$ is a *shunt resistor* used to produce a small voltage drop for the purpose of sensing current in a circuit. Such shunt resistors must be low-value in order not to impose an undue load on the rest of the circuit. An example of a necessary resistor value much greater than $100\text{ k}\Omega$ is an electrostatic *drain resistor* used to dissipate stored electric charges from body capacitance for the sake of preventing damage to sensitive semiconductor components, while also preventing a path for current that could be dangerous to the person (i.e. shock).

⁵Admittedly, solderless breadboards are very useful for constructing complex electronic circuits with many components, especially DIP-style integrated circuits (ICs), but they tend to give trouble with connection integrity after frequent use. An alternative for projects using low counts of ICs is to solder IC sockets into prototype printed circuit boards (PCBs) and run wires from the soldered pins of the IC sockets to terminal blocks where reliable temporary connections may be made.

7.1.3 Terminal blocks for circuit construction

Terminal blocks are the standard means for making electric circuit connections in industrial systems. They are also quite useful as a learning tool, and so I highly recommend their use in lieu of solderless breadboards⁶. Terminal blocks provide highly reliable connections capable of withstanding significant voltage and current magnitudes, and they force the builder to think very carefully about component layout which is an important mental practice. Terminal blocks that mount on standard 35 mm DIN rail⁷ are made in a wide range of types and sizes, some with built-in disconnecting switches, some with built-in components such as rectifying diodes and fuseholders, all of which facilitate practical circuit construction.

I recommend every student of electricity build their own terminal block array for use in constructing experimental circuits, consisting of several terminal blocks where each block has at least 4 connection points all electrically common to each other⁸ and at least one terminal block that is a fuse holder for overcurrent protection. A pair of anchoring blocks hold all terminal blocks securely on the DIN rail, preventing them from sliding off the rail. Each of the terminals should bear a number, starting from 0. An example is shown in the following photograph and illustration:



Screwless terminal blocks (using internal spring clips to clamp wire and component lead ends) are preferred over screw-based terminal blocks, as they reduce assembly and disassembly time, and also minimize repetitive wrist stress from twisting screwdrivers. Some screwless terminal blocks require the use of a special tool to release the spring clip, while others provide buttons⁹ for this task which may be pressed using the tip of any suitable tool.

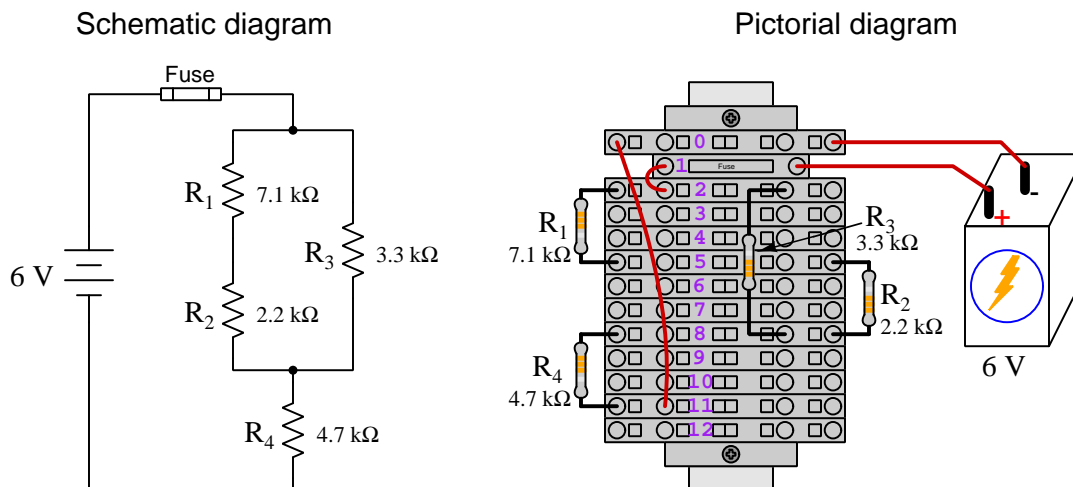
⁶Solderless breadboard are preferable for complicated electronic circuits with multiple integrated “chip” components, but for simpler circuits I find terminal blocks much more practical. An alternative to solderless breadboards for “chip” circuits is to solder chip sockets onto a PCB and then use wires to connect the socket pins to terminal blocks. This also accommodates *surface-mount* components, which solderless breadboards do not.

⁷DIN rail is a metal rail designed to serve as a mounting point for a wide range of electrical and electronic devices such as terminal blocks, fuses, circuit breakers, relay sockets, power supplies, data acquisition hardware, etc.

⁸Sometimes referred to as *equipotential*, *same-potential*, or *potential distribution* terminal blocks.

⁹The small orange-colored squares seen in the above photograph are buttons for this purpose, and may be actuated by pressing with any tool of suitable size.

The following example shows how such a terminal block array might be used to construct a series-parallel resistor circuit consisting of four resistors and a battery:



Numbering on the terminal blocks provides a very natural translation to SPICE¹⁰ netlists, where component connections are identified by terminal number:

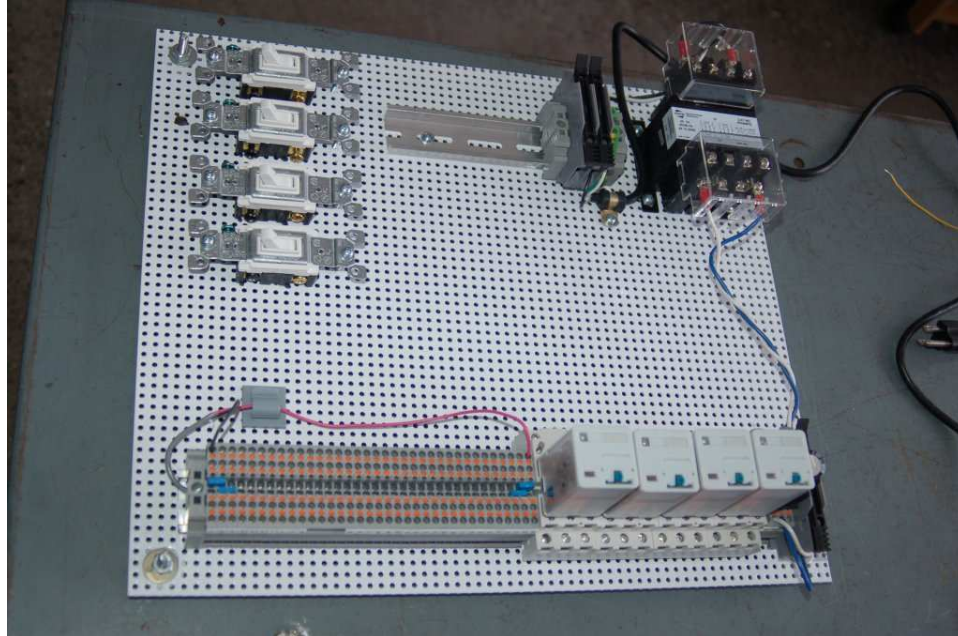
```
* Series-parallel resistor circuit
v1 1 0 dc 6
r1 2 5 7100
r2 5 8 2200
r3 2 8 3300
r4 8 11 4700
rjmp1 1 2 0.01
rjmp2 0 11 0.01
.op
.end
```

Note the use of “jumper” resistances `rjmp1` and `rjmp2` to describe the wire connections between terminals 1 and 2 and between terminals 0 and 11, respectively. Being resistances, SPICE requires a resistance value for each, and here we see they have both been set to an arbitrarily low value of 0.01 Ohm realistic for short pieces of wire.

Listing all components and wires along with their numbered terminals happens to be a useful documentation method for any circuit built on terminal blocks, independent of SPICE. Such a “wiring sequence” may be thought of as a *non-graphical description* of an electric circuit, and is exceptionally easy to follow.

¹⁰SPICE is computer software designed to analyze electrical and electronic circuits. Circuits are described for the computer in the form of *netlists* which are text files listing each component type, connection node numbers, and component values.

An example of a more elaborate terminal block array is shown in the following photograph, with terminal blocks and “ice-cube” style electromechanical relays mounted to DIN rail, which is turn mounted to a perforated subpanel¹¹. This “terminal block board” hosts an array of thirty five undedicated terminal block sections, four SPDT toggle switches, four DPDT “ice-cube” relays, a step-down control power transformer, bridge rectifier and filtering capacitor, and several fuses for overcurrent protection:



Four plastic-bottomed “feet” support the subpanel above the benchtop surface, and an unused section of DIN rail stands ready to accept other components. Safety features include electrical bonding of the AC line power cord’s ground to the metal subpanel (and all metal DIN rails), mechanical strain relief for the power cord to isolate any cord tension from wire connections, clear plastic finger guards covering the transformer’s screw terminals, as well as fused overcurrent protection for the 120 Volt AC line power and the transformer’s 12 Volt AC output. The perforated holes happen to be on $\frac{1}{4}$ inch centers, their presence making it very easy to attach other sections of DIN rail, or specialized electrical components, directly to the grounded metal subpanel. Such a “terminal block board” is an inexpensive¹² yet highly flexible means to construct physically robust circuits using industrial wiring practices.

¹¹An electrical *subpanel* is a thin metal plate intended for mounting inside an electrical enclosure. Components are attached to the subpanel, and the subpanel in turn bolts inside the enclosure. Subpanels allow circuit construction outside the confines of the enclosure, which speeds assembly. In this particular usage there is no enclosure, as the subpanel is intended to be used as an open platform for the convenient construction of circuits on a benchtop by students. In essence, this is a modern version of the traditional *breadboard* which was literally a wooden board such as might be used for cutting loaves of bread, but which early electrical and electronic hobbyists used as platforms for the construction of circuits.

¹²At the time of this writing (2019) the cost to build this board is approximately \$250 US dollars.

7.1.4 Conducting experiments

An *experiment* is an exploratory act, a test performed for the purpose of assessing some proposition or principle. Experiments are the foundation of the *scientific method*, a process by which careful observation helps guard against errors of speculation. All good experiments begin with an *hypothesis*, defined by the American Heritage Dictionary of the English Language as:

An assertion subject to verification or proof, as (a) A proposition stated as a basis for argument or reasoning. (b) A premise from which a conclusion is drawn. (c) A conjecture that accounts, within a theory or ideational framework, for a set of facts and that can be used as a basis for further investigation.

Stated plainly, an hypothesis is an *educated guess* about cause and effect. The correctness of this initial guess matters little, because any well-designed experiment will reveal the truth of the matter. In fact, *incorrect* hypotheses are often the most valuable because the experiments they engender lead us to surprising discoveries. One of the beautiful aspects of science is that it is more focused on the process of *learning* than about the status of *being correct*¹³. In order for an hypothesis to be valid, it must be testable¹⁴, which means it must be a claim possible to refute given the right data. Hypotheses impossible to critique are useless.

Once an hypothesis has been formulated, an experiment must be designed to test that hypothesis. A well-designed experiment requires careful regulation of all relevant variables, both for personal safety and for prompting the hypothesized results. If the effects of one particular variable are to be tested, the experiment must be run multiple times with different values of (only) that particular variable. The experiment set up with the “baseline” variable set is called the *control*, while the experiment set up with different value(s) is called the *test* or *experimental*.

For some hypotheses a viable alternative to a physical experiment is a *computer-simulated experiment* or even a *thought experiment*. Simulations performed on a computer test the hypothesis against the physical laws encoded within the computer simulation software, and are particularly useful for students learning new principles for which simulation software is readily available¹⁵.

¹³Science is more about clarifying our view of the universe through a systematic process of error detection than it is about proving oneself to be right. Some *scientists* may happen to have large egos – and this may have more to do with the ways in which large-scale scientific research is *funded* than anything else – but *scientific method* itself is devoid of ego, and if embraced as a practical philosophy is quite an effective stimulant for humility. Within the education system, scientific method is particularly valuable for helping students break free of the crippling fear of *being wrong*. So much emphasis is placed in formal education on assessing correct retention of facts that many students are fearful of saying or doing anything that might be perceived as a mistake, and of course making mistakes (i.e. having one’s hypotheses disproven by experiment) is an indispensable tool for learning. Introducing science in the classroom – *real* science characterized by individuals forming actual hypotheses and testing those hypotheses by experiment – helps students become self-directed learners.

¹⁴This is the principle of *falsifiability*: that a scientific statement has value only insofar as it is liable to disprove given the requisite experimental evidence. Any claim that is unfalsifiable – that is, a claim which can *never* be disproven by any evidence whatsoever – could be completely wrong and we could never know it.

¹⁵A very pertinent example of this is learning how to analyze electric circuits using simulation software such as SPICE. A typical experimental cycle would proceed as follows: (1) Find or invent a circuit to analyze; (2) Apply your analytical knowledge to that circuit, predicting all voltages, currents, powers, etc. relevant to the concepts you are striving to master; (3) Run a simulation on that circuit, collecting “data” from the computer when complete; (4) Evaluate whether or not your hypotheses (i.e. predicted voltages, currents, etc.) agree with the computer-generated results; (5) If so, your analyses are (provisionally) correct – if not, examine your analyses and the computer simulation again to determine the source of error; (6) Repeat this process as many times as necessary until you achieve mastery.

Thought experiments are useful for detecting inconsistencies within your own understanding of some subject, rather than testing your understanding against physical reality.

Here are some general guidelines for conducting experiments:

- The clearer and more specific the hypothesis, the better. Vague or unfalsifiable hypotheses are useless because they will fit *any* experimental results, and therefore the experiment cannot teach you anything about the hypothesis.
- Collect as much data (i.e. information, measurements, sensory experiences) generated by an experiment as is practical. This includes the time and date of the experiment, too!
- *Never* discard or modify data gathered from an experiment. If you have reason to believe the data is unreliable, write notes to that effect, but never throw away data just because you think it is untrustworthy. It is quite possible that even “bad” data holds useful information, and that someone else may be able to uncover its value even if you do not.
- Prioritize *quantitative* data over *qualitative* data wherever practical. Quantitative data is more specific than qualitative, less prone to subjective interpretation on the part of the experimenter, and amenable to an arsenal of analytical methods (e.g. statistics).
- Guard against your own bias(es) by making your experimental results available to others. This allows other people to scrutinize your experimental design and collected data, for the purpose of detecting and correcting errors you may have missed. Document your experiment such that others may independently replicate it.
- Always be looking for sources of error. No physical measurement is perfect, and so it is impossible to achieve *exact* values for any variable. Quantify the amount of uncertainty (i.e. the “tolerance” of errors) whenever possible, and be sure your hypothesis does not depend on precision better than this!
- Always remember that scientific confirmation is provisional – no number of “successful” experiments will prove an hypothesis true for all time, but a single experiment can disprove it. Put into simpler terms, *truth is elusive but error is within reach*.
- Remember that scientific method is about *learning*, first and foremost. An unfortunate consequence of scientific triumph in modern society is that science is often viewed by non-practitioners as an unerring source of truth, when in fact science is an ongoing process of challenging existing ideas to probe for errors and oversights. This is why it is perfectly acceptable to have a failed hypothesis, and why the only truly failed experiment is one where nothing was learned.

The following is an example of a well-planned and executed experiment, in this case a physical experiment demonstrating Ohm's Law.

Planning Time/Date = 09:30 on 12 February 2019

HYPOTHESIS: the current through any resistor should be exactly proportional to the voltage impressed across it.

PROCEDURE: connect a resistor rated 1 k Ohm and 1/4 Watt to a variable-voltage DC power supply. Use an ammeter in series to measure resistor current and a voltmeter in parallel to measure resistor voltage.

RISKS AND MITIGATION: excessive power dissipation may harm the resistor and/or pose a burn hazard, while excessive voltage poses an electric shock hazard. 30 Volts is a safe maximum voltage for laboratory practices, and according to Joule's Law a 1000 Ohm resistor will dissipate 0.25 Watts at 15.81 Volts ($P = V^2 / R$), so I will remain below 15 Volts just to be safe.

Experiment Time/Date = 10:15 on 12 February 2019

DATA COLLECTED:

(Voltage)	(Current)	(Voltage)	(Current)
0.000 V	= 0.000 mA	8.100	= 7.812 mA
2.700 V	= 2.603 mA	10.00 V	= 9.643 mA
5.400 V	= 5.206 mA	14.00 V	= 13.49 mA

Analysis Time/Date = 10:57 on 12 February 2019

ANALYSIS: current definitely increases with voltage, and although I expected exactly one milliAmpere per Volt the actual current was usually less than that. The voltage/current ratios ranged from a low of 1036.87 (at 8.1 Volts) to a high of 1037.81 (at 14 Volts), but this represents a variance of only -0.0365% to +0.0541% from the average, indicating a very consistent proportionality -- results consistent with Ohm's Law.

ERROR SOURCES: one major source of error is the resistor's value itself. I did not measure it, but simply assumed color bands of brown-black-red meant exactly 1000 Ohms. Based on the data I think the true resistance is closer to 1037 Ohms. Another possible explanation is multimeter calibration error. However, neither explains the small positive and negative variances from the average. This might be due to electrical noise, a good test being to repeat the same experiment to see if the variances are the same or different. Noise should generate slightly different results every time.

The following is an example of a well-planned and executed *virtual* experiment, in this case demonstrating Ohm's Law using a computer (SPICE) simulation.

Planning Time/Date = 12:32 on 14 February 2019

HYPOTHESIS: for any given resistor, the current through that resistor should be exactly proportional to the voltage impressed across it.

PROCEDURE: write a SPICE netlist with a single DC voltage source and single 1000 Ohm resistor, then use NGSPICE version 26 to perform a "sweep" analysis from 0 Volts to 25 Volts in 5 Volt increments.

```
* SPICE circuit
v1 1 0 dc
r1 1 0 1000
.dc v1 0 25 5
.print dc v(1) i(v1)
.end
```

RISKS AND MITIGATION: none.

DATA COLLECTED:

DC transfer characteristic Thu Feb 14 13:05:08 2019

Index	v-sweep	v(1)	v1#branch
0	0.000000e+00	0.000000e+00	0.000000e+00
1	5.000000e+00	5.000000e+00	-5.00000e-03
2	1.000000e+01	1.000000e+01	-1.00000e-02
3	1.500000e+01	1.500000e+01	-1.50000e-02
4	2.000000e+01	2.000000e+01	-2.00000e-02
5	2.500000e+01	2.500000e+01	-2.50000e-02

Analysis Time/Date = 13:06 on 14 February 2019

ANALYSIS: perfect agreement between data and hypothesis -- current is precisely 1/1000 of the applied voltage for all values. Anything other than perfect agreement would have probably meant my netlist was incorrect. The negative current values surprised me, but it seems this is just how SPICE interprets normal current through a DC voltage source.

ERROR SOURCES: none.

As gratuitous as it may seem to perform experiments on a physical law as well-established as Ohm's Law, even the examples listed previously demonstrate opportunity for real learning. In the physical experiment example, the student should identify and explain why their data does not perfectly agree with the hypothesis, and this leads them naturally to consider sources of error. In the computer-simulated experiment, the student is struck by SPICE's convention of denoting regular current through a DC voltage source as being *negative* in sign, and this is also useful knowledge for future simulations. Scientific experiments are most interesting when things *do not* go as planned!

Aside from verifying well-established physical laws, simple experiments are extremely useful as educational tools for a wide range of purposes, including:

- Component familiarization (e.g. *Which terminals of this switch connect to the NO versus NC contacts?*)
- System testing (e.g. *How heavy of a load can my AC-DC power supply source before the semiconductor components reach their thermal limits?*)
- Learning programming languages (e.g. *Let's try to set up an "up" counter function in this PLC!*)

Above all, the priority here is to inculcate the habit of hypothesizing, running experiments, and analyzing the results. This experimental cycle not only serves as an excellent method for self-directed learning, but it also works exceptionally well for troubleshooting faults in complex systems, and for these reasons should be a part of every technician's and every engineer's education.

7.1.5 Constructing projects

Designing, constructing, and testing projects is a very effective means of practical education. Within a formal educational setting, projects are generally chosen (or at least vetted) by an instructor to ensure they may be reasonably completed within the allotted time of a course or program of study, and that they sufficiently challenge the student to learn certain important principles. In a self-directed environment, projects are just as useful as a learning tool but there is some risk of unwittingly choosing a project beyond one's abilities, which can lead to frustration.

Here are some general guidelines for managing projects:

- Define your goal(s) before beginning a project: what do you wish to achieve in building it? What, exactly, should the completed project *do*?
- Analyze your project prior to construction. Document it in appropriate forms (e.g. schematic diagrams), predict its functionality, anticipate all associated risks. In other words, *plan ahead*.
- Set a reasonable budget for your project, and stay within it.
- Identify any deadlines, and set reasonable goals to meet those deadlines.
- Beware of *scope creep*: the tendency to modify the project's goals before it is complete.
- Document your progress! An easy way to do this is to use photography or videography: take photos and/or videos of your project as it progresses. Document failures as well as successes, because both are equally valuable from the perspective of learning.

7.2 Experiment: (first experiment)

Conduct an experiment to . . .

EXPERIMENT CHECKLIST:

- Prior to experimentation:
 - Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.
 - Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).
 - Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.
- During experimentation:
 - Safe practices followed at all times (e.g. no contact with energized circuit).
 - Correct equipment usage according to manufacturer's recommendations.
 - All data collected, ideally quantitative with full precision (i.e. no rounding).
- After each experimental run:
 - If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.
 - Identify any uncontrolled sources of error in the experiment.
- After all experimental re-runs:
 - Save all data for future reference.
 - Write an analysis of experimental results and lessons learned.

Challenges

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- ???.
- ???.

7.3 Project: (first project)

This is a description of the project!

PROJECT CHECKLIST:

- Prior to construction:
 - Prototype diagram(s) and description of project scope.
 - Risk assessment/mitigation plan.
 - Timeline and action plan.
- During construction:
 - Safe work habits (e.g. no contact made with energized circuit at any time).
 - Correct equipment usage according to manufacturer's recommendations.
 - Timeline and action plan amended as necessary.
 - Maintain the originally-planned project scope (i.e. avoid adding features!).
- After completion:
 - All functions tested against original plan.
 - Full, accurate, and appropriate documentation of all project details.
 - Complete bill of materials.
 - Written summary of lessons learned.

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

“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.

The following principles inform the instructional and assessment philosophies embodied in these learning modules:

- 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 environment¹ 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 Socratic² 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 absurdity³ 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

¹In 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.

²Socrates 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.

³This 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 characteristic of these learning modules is a lack of step-by-step instructions in the Project and Experiment chapters. Unlike many modern workbooks and laboratory guides where step-by-step instructions are prescribed for each experiment, these modules take the approach that students must learn to closely read the tutorials and apply their own reasoning to identify the appropriate experimental steps. Sometimes these steps are plainly declared in the text, just not as a set of enumerated points. At other times certain steps are implied, an example being assumed competence in test equipment use where the student should not need to be told *again* how to use their multimeter because that was thoroughly explained in previous lessons. In some circumstances no steps are given at all, leaving the entire procedure up to the student.

This lack of prescription is not a flaw, but rather a feature. Close reading and clear thinking are foundational principles of this learning series, and in keeping with this philosophy all activities are designed to *require* those behaviors. Some students may find the lack of prescription frustrating, because it demands more from them than what their previous educational experiences required. This frustration should be interpreted as an unfamiliarity with autonomous thinking, a problem which must be corrected if the student is ever to become a self-directed learner and effective problem-solver. Ultimately, the need for students to read closely and think clearly is more important both in the near-term and far-term than any specific facet of the subject matter at hand. If a student takes longer than expected to complete a module because they are forced to outline, digest, and reason on their own, so be it. The future gains enjoyed by developing this mental discipline will be well worth the additional effort and delay.

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.

This methodology of assigning text-based modules to students for digestion and then using Socratic dialogue to assess progress and hone students’ thinking was developed over a period of several years by the author with his Electronics and Instrumentation students at the two-year college level. While decidedly unconventional and sometimes even unsettling for students accustomed to a more passive lecture environment, this instructional philosophy has proven its ability to convey conceptual mastery, foster careful analysis, and enhance employability so much better than lecture that the author refuses to ever teach by lecture again.

Problems which often go undiagnosed in a lecture environment are laid bare in this “inverted” format where students must articulate and logically defend their reasoning. This, too, may be unsettling for students accustomed to lecture sessions where the instructor cannot tell for sure who comprehends and who does not, and this vulnerability necessitates sensitivity on the part of the “inverted” session instructor in order that students never feel discouraged by having their errors exposed. *Everyone* makes mistakes from time to time, and learning is a lifelong process! Part of the instructor’s job is to build a culture of learning among the students where errors are not seen as shameful, but rather as opportunities for progress.

To this end, instructors managing courses based on these modules should adhere to the following principles:

- Student questions are always welcome and demand thorough, honest answers. The only type of question an instructor should refuse to answer is one the student should be able to easily answer on their own. Remember, *the fundamental goal of education is for each student to learn to think clearly and independently*. This requires hard work on the part of the student, which no instructor should ever circumvent. Anything done to bypass the student's responsibility to do that hard work ultimately limits that student's potential and thereby does real harm.
- It is not only permissible, but encouraged, to answer a student's question by asking questions in return, these follow-up questions designed to guide the student to reach a correct answer through their own reasoning.
- All student answers demand to be challenged by the instructor and/or by other students. This includes both correct and incorrect answers – the goal is to practice the articulation and defense of one's own reasoning.
- No reading assignment is deemed complete unless and until the student demonstrates their ability to accurately summarize the major points in their own terms. Recitation of the original text is unacceptable. This is why every module contains an "Outline and reflections" question as well as a "Foundational concepts" question in the Conceptual reasoning section, to prompt reflective reading.
- No assigned question is deemed answered unless and until the student demonstrates their ability to consistently and correctly apply the concepts to *variations* of that question. This is why module questions typically contain multiple "Challenges" suggesting different applications of the concept(s) as well as variations on the same theme(s). Instructors are encouraged to devise as many of their own "Challenges" as they are able, in order to have a multitude of ways ready to probe students' understanding.
- No assigned experiment or project is deemed complete unless and until the student demonstrates the task in action. If this cannot be done "live" before the instructor, video-recordings showing the demonstration are acceptable. All relevant safety precautions must be followed, all test equipment must be used correctly, and the student must be able to properly explain all results. The student must also successfully answer all Challenges presented by the instructor for that experiment or project.

Students learning from these modules would do well to abide by the following principles:

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples*.
- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.
- Never take the easy path of highlighting or underlining important text. Instead, *summarize* and/or *comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.
- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.
- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.
- 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 varied⁴ effort, and never give up!

Students interested in using these modules for self-study will also find them beneficial, although the onus of responsibility for thoroughly reading and answering questions will of course lie with that individual alone. If a qualified instructor is not available to challenge students, a workable alternative is for students to form study groups where they challenge⁵ one another.

To high standards of education,

Tony R. Kuphaldt

⁴As 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.

⁵Avoid the temptation to simply share answers with study partners, as this is really counter-productive to learning. Always bear in mind that the answer to any question is far less important in the long run than the method(s) used to obtain that answer. The goal of education is to empower one’s life through the improvement of clear and independent thought, literacy, expression, and various practical skills.

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 a loaf of 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.

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.

9 January 2020 – continued writing a Programming Reference section.

8 January 2020 – began writing a Programming Reference chapter, including a section showing a simple common-source amplifier modeled using C++.

23 October 2019 – document first created.

Index

- AC gain, 6
- Adding quantities to a qualitative problem, 60
- Amplification, 5
- Annotating diagrams, 59

- Bias, 6
- Breadboard, solderless, 48, 49
- Breadboard, traditional, 51

- C++, 14
- Cardio-Pulmonary Resuscitation, 46
- Checking for exceptions, 60
- Checking your work, 60
- Class-A amplifier operation, 22
- Class-B amplifier operation, 22
- Class-C amplifier operation, 22
- Code, computer, 67
- Compiler, C++, 14
- Computer programming, 13
- CPR, 46

- Dalziel, Charles, 46
- Dimensional analysis, 59
- DIN rail, 49
- DIP, 48

- Edwards, Tim, 68
- Electric shock, 46
- Electrically common points, 47
- Enclosure, electrical, 51
- Equipotential points, 47, 49
- Experiment, 52
- Experimental guidelines, 53

- Gain, 6
- Graph values to solve a problem, 60
- Greenleaf, Cynthia, 31

- How to teach with these modules, 62
- Hwang, Andrew D., 69

- IC, 48
- Identify given data, 59
- Identify relevant principles, 59
- Instructions for projects and experiments, 63
- Intermediate results, 59
- Interpreter, Python, 18
- Inverted instruction, 62

- Java, 15

- Knuth, Donald, 68

- Lamport, Leslie, 68
- Limiting cases, 60

- Maxwell, James Clerk, 7
- Metacognition, 36
- Moolenaar, Bram, 67
- Murphy, Lynn, 31

- Open-source, 67

- Potential distribution, 49
- Problem-solving: annotate diagrams, 59
- Problem-solving: check for exceptions, 60
- Problem-solving: checking work, 60
- Problem-solving: dimensional analysis, 59
- Problem-solving: graph values, 60
- Problem-solving: identify given data, 59
- Problem-solving: identify relevant principles, 59
- Problem-solving: interpret intermediate results, 59
- Problem-solving: limiting cases, 60
- Problem-solving: qualitative to quantitative, 60
- Problem-solving: quantitative to qualitative, 60

- Problem-solving: reductio ad absurdum, 60
- Problem-solving: simplify the system, 59
- Problem-solving: thought experiment, 53, 59
- Problem-solving: track units of measurement, 59
- Problem-solving: visually represent the system, 59
- Problem-solving: work in reverse, 60
- Programming, computer, 13
- Project management guidelines, 56
- Python, 18

- Qualitatively approaching a quantitative problem, 60

- Reading Apprenticeship, 31
- Reductio ad absurdum, 60–62

- Safety, electrical, 46
- Schoenbach, Ruth, 31
- Scientific method, 36, 52
- Scope creep, 56
- Shunt resistor, 48
- Simplifying a system, 59
- Socrates, 61
- Socratic dialogue, 62
- Solderless breadboard, 48, 49
- Source code, 14
- SPICE, 31, 53
- SPICE netlist, 50
- Stallman, Richard, 67
- Subpanel, 51
- Surface mount, 49

- Terminal block, 47–51
- Thought experiment, 53, 59
- Torvalds, Linus, 67
- Transistor, 5

- Units of measurement, 59

- Vacuum tube, 5
- Visualizing a system, 59

- Whitespace, C++, 14, 15
- Whitespace, Python, 21
- Wiring sequence, 50
- Work in reverse to solve a problem, 60
- WYSIWYG, 67, 68