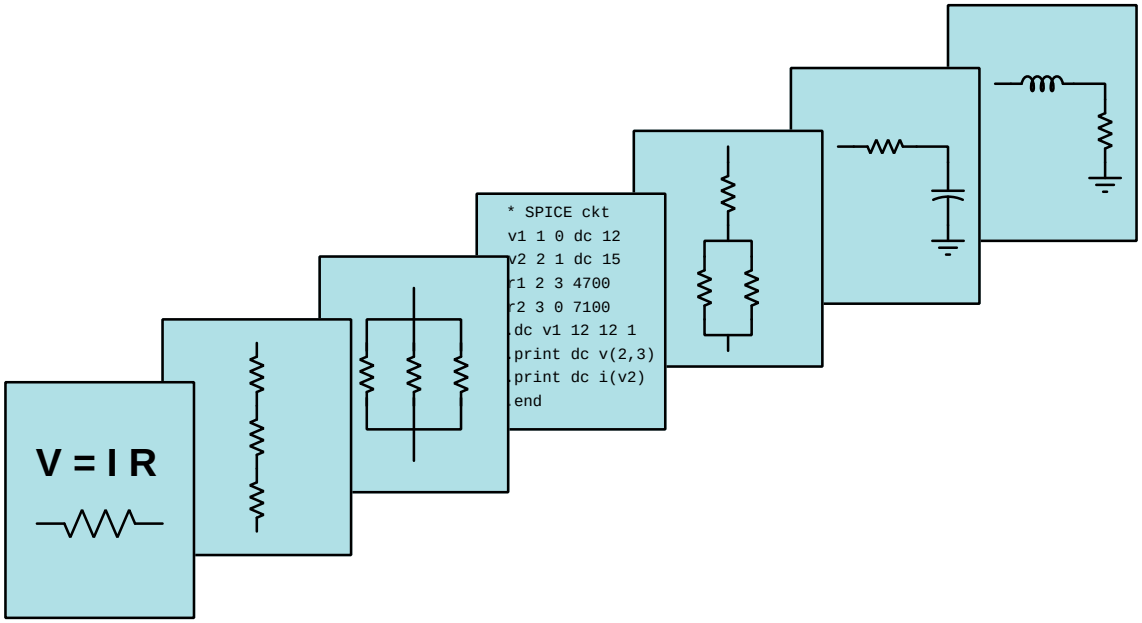
Modular Electronics Learning (ModEL) project



PID Control

Last update = 18 February 2025

Contents

Chapter 1

Introduction

Chapter 2

Tutorial

2.1 Process control terms and definitions

The field of industrial measurement and control has its own unique terms and standards, some common terms defined below:

Process – The physical system we are attempting to control or measure. *Examples: water filtration system, molten metal casting system, steam boiler, oil refinery unit, power generation unit, vehicle speed, robot arm position.*

Process Variable, or **PV** – The specific quantity we are measuring in a process. *Examples: fluid pressure, liquid level, temperature, fluid flow, electrical conductivity, water pH, machine position, shaft rotational speed, vibration.*

Setpoint, or **SP** – The value at which we desire the process variable to be maintained at. In other words, the "target" value for the process variable.

Primary Sensing Element, or **PSE** – A device directly sensing the process variable and translating that sensed quantity into an analog representation (electrical voltage, current, resistance; mechanical force, motion, etc.). *Examples: thermocouple, thermistor, bourdon tube, microphone, potentiometer, electrochemical cell, accelerometer.*

Transmitter – A device translating the signal produced by a primary sensing element (PSE) into a *standardized* electronic signal that a control system can understand.

Controller – A device receiving a process variable (PV) signal from a primary sensing element (PSE) or transmitter, comparing that signal to the desired value (called the setpoint) for that process variable, and calculating an appropriate output signal value to be sent to a final control element (FCE) such as an electric motor or control valve.

Final Control Element, or **FCE** – A device receiving the signal output by a controller to directly influence the process. *Examples: variable-speed electric motor, control valve, electric heater.*

Manipulated Variable, or **MV** – The quantity in a process we adjust or otherwise manipulate in order to influence the process variable (PV). Also used to describe the output signal generated by a controller; i.e. the signal commanding ("manipulating") the final control element to influence the process.

Load – any uncontrolled factor affecting the process variable's value. *Example: a window opening in a room letting warm air out and cold air in, affecting that room's temperature in such a way that that thermostatic heating system must compensate to maintain the room's temperature at setpoint.*

Automatic mode – When the controller generates an output signal based on the relationship of process variable (PV) to the setpoint (SP).

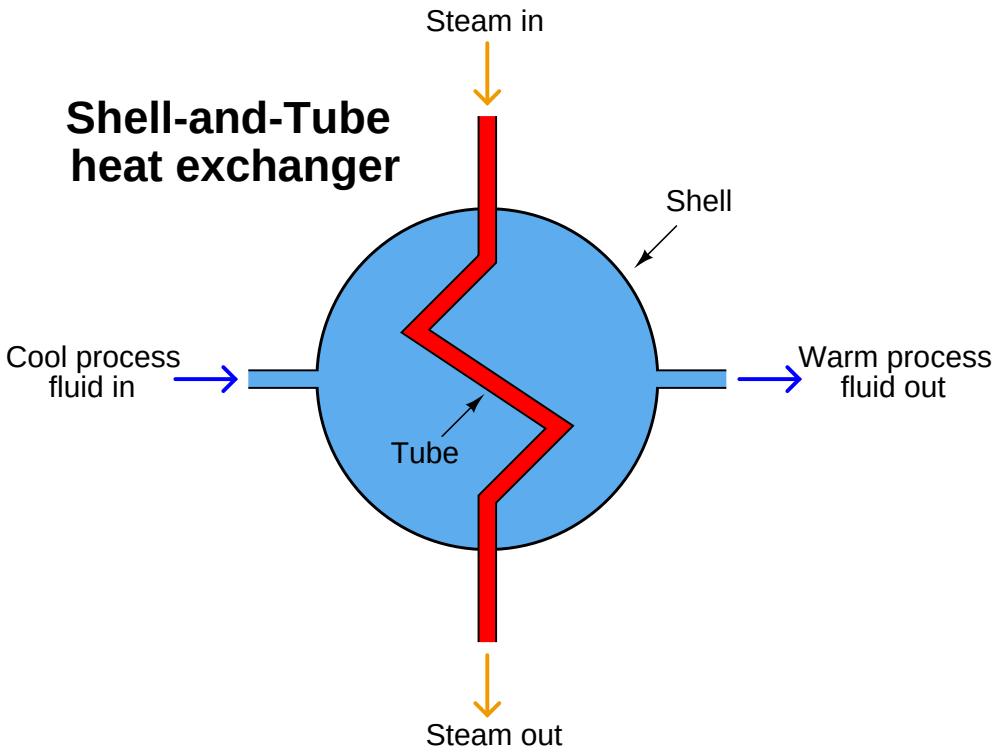
Manual mode – When the controller's decision-making ability is bypassed to let a human operator directly determine the output signal sent to the final control element.

Loop – This widely-used term unfortunately has multiple meanings. In one sense it refers to the

complete electrical circuit comprising a 4-20 mA analog measurement or control signal. In another sense it refers to the circular flow of information in any negative feedback regulation system.

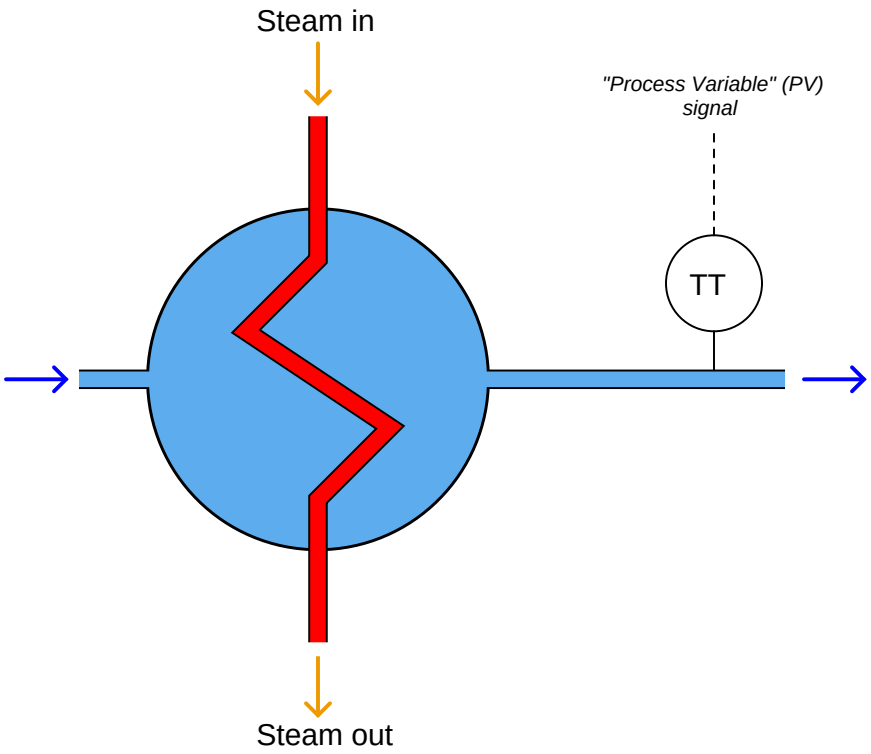
2.2 Basic feedback control principles

Before we begin our discussion on process control, we must define a few key terms. First, we have what is known as the *process*: the physical system we wish to monitor and control. For the sake of illustration, consider a heat exchanger that uses high-temperature steam to transfer heat to a lower-temperature liquid. Heat exchangers are used frequently in the chemical industries to maintain the necessary temperature of a chemical solution, so the desired blending, separation, or reactions can occur. A very common design of heat exchanger is the "shell-and-tube" style, where a metal shell serves as a conduit for the chemical solution to flow through, while a network of smaller tubes runs through the interior of the shell, carrying steam or some other heat-transfer fluid. The hotter steam flowing through the tubes transfers heat energy to the cooler process fluid surrounding the tubes, inside the shell of the heat exchanger:



In this case, the *process* is the entire heating system, consisting of the fluid we wish to heat, the heat exchanger, and the steam delivering the required heat energy. In order to maintain steady control of the process fluid's exiting temperature, we must find a way to measure it and represent that measurement in signal form so it may be interpreted by other instruments taking some form of control action. In instrumentation terms, the measuring device is known as a *transmitter*, because it *transmits* the process measurement in the form of a signal.

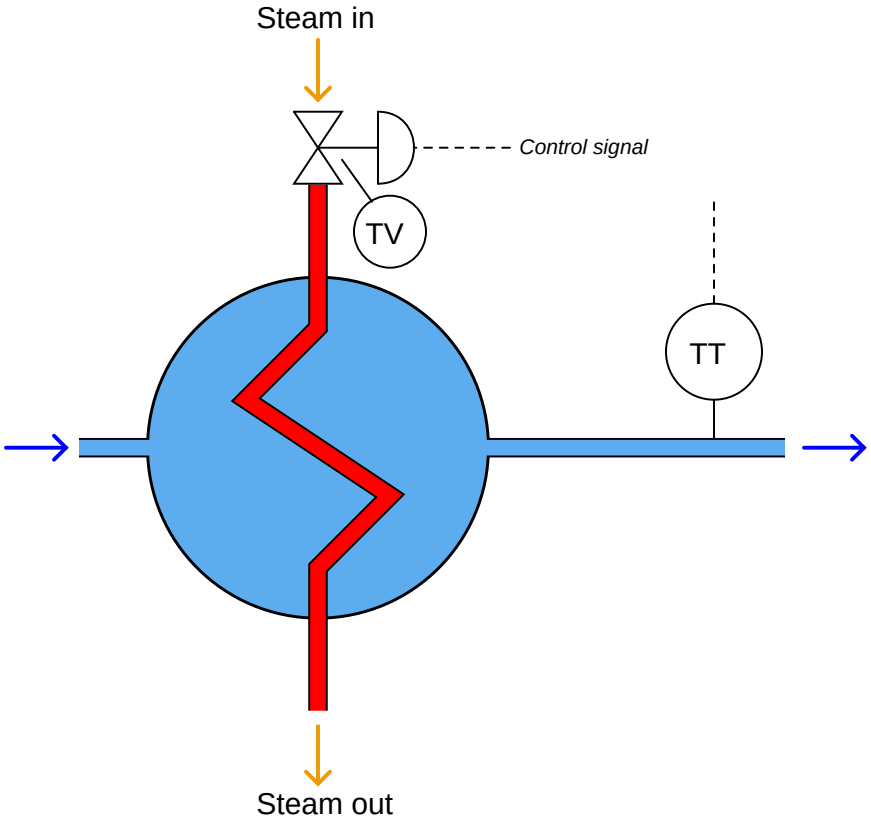
Transmitters are represented in process diagrams by small circles with identifying letters inside, in this case, "TT," which stands for **T**emperature **T**ransmitter:



The signal output by the transmitter (represented by the "PV" dashed line), representing the heated fluid's exiting temperature, is called the *process variable*. Like a variable in a mathematical equation that represents some story-problem quantity, this signal represents the measured quantity we wish to control in the process.

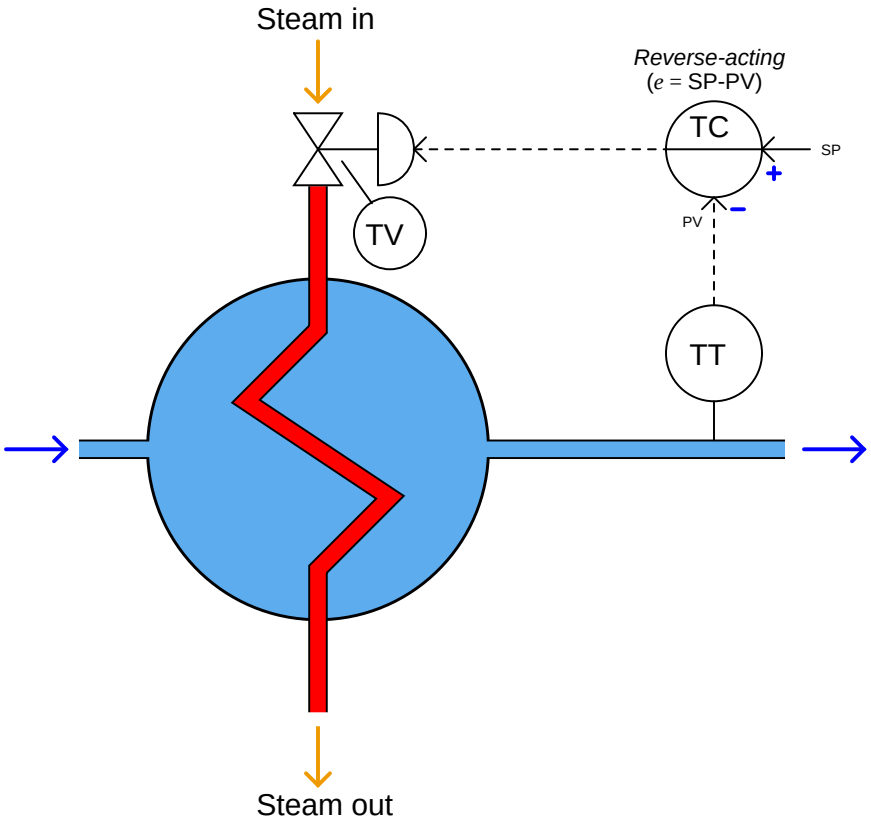
In order to exert control over the process variable, we must have some way of altering fluid flow through the heat exchanger, either of the process fluid, the steam, or both. Generally, it makes more sense to alter the flow of the heating medium (the steam), and let the process fluid flow rate be dictated by the demands of the larger process. If this heat exchanger were part of an oil refinery unit, for example, it would be far better to throttle steam flow to control oil temperature rather than to throttle the oil flow itself, since altering the oil's flow will also affect other process variables upstream and downstream of the exchanger. Ideally, the heat exchanger temperature control system would provide consistent temperature of the exiting oil, for any given incoming oil temperature and flow-rate of oil through it.

One convenient way to throttle steam flow into the heat exchanger is to use a control valve (labeled "TV" because it is a **T**emperature **V**alve). In general terms, a control valve is known as a *final control element*. Other types of final control elements exist (servo motors, variable-flow pumps, and other mechanical devices used to vary some physical quantity at will), but valves are the most common, and probably the simplest to understand. With a final control element in place, the steam flow becomes known as the *manipulated variable*, because it is the quantity we will manipulate in order to gain control over the process variable:



Valves come in a wide variety of sizes and styles. Some valves are hand-operated: that is, they have a "wheel" or other form of manual control that may be moved to "pinch off" or "open up" the flow passage through the pipe. Other valves come equipped with signal receivers and positioner devices, which move the valve mechanism to various positions at the command of a signal (usually an electrical signal, like the type output by transmitter instruments). This feature allows for remote control, so a human operator or computer device may exert control over the manipulated variable from a distance. In the previous illustration, the steam control valve is equipped with such an electrical signal input, represented by the "control signal" dashed line.

This brings us to the final component of the heat exchanger temperature control system: the *controller*. This is a device designed to interpret the transmitter's process variable signal and decide how far open the control valve needs to be in order to maintain that process variable at the desired value.



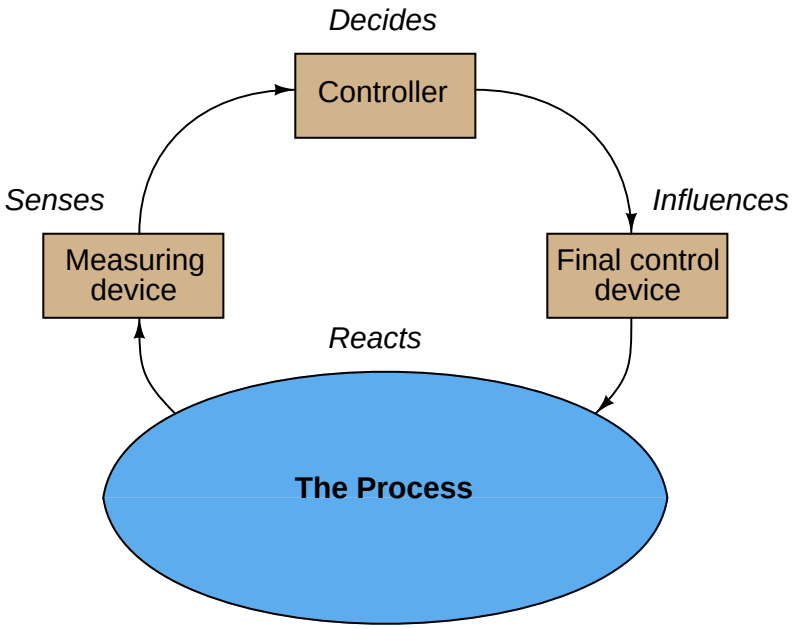
Here, the circle with the letters "TC" in the center represents the controller. Those letters stand for **T**emperature **C**ontroller, since the process variable being controlled is the process fluid's *temperature*. Usually, the controller consists of a computer making automatic decisions to open and close the valve as necessary to stabilize the process variable at some predetermined *setpoint*.

Note that the controller's circle has a solid line going through the center of it, while the transmitter and control valve circles are open. An open circle represents a field-mounted device according to the ISA standard for instrumentation symbols, and a single solid line through the middle of a circle tells us the device is located on the front of a control panel in a main control room location. So, even though the diagram might appear as though these three instruments are located close to one another, they in fact may be quite far apart. Both the transmitter and the valve must be located near the heat exchanger (out in the "field" area rather than inside a building), but the controller may be located a long distance away where human operators can adjust the setpoint from inside a safe and secure control room.

These elements comprise the essentials of a *feedback control system*: the *process* (the system

to be controlled), the *process variable* (the specific quantity to be measured and controlled), the *transmitter* (the device used to measure the process variable and output a corresponding signal), the *controller* (the device that decides what to do to bring the process variable as close to setpoint as possible), the *final control element* (the device that directly exerts control over the process), and the *manipulated variable* (the quantity to be directly altered to effect control over the process variable).

Feedback control may be viewed as a sort of information "loop," from the transmitter (measuring the process variable), to the controller, to the final control element, and through the process itself, back to the transmitter. Ideally, a process control "loop" not only holds the process variable at a steady level (the setpoint), but also maintains control over the process variable given changes in setpoint, and even changes in other variables of the process:



Specifically, the type of feedback we are employing here to control the process is *negative* or *degenerative* feedback. The term "negative" refers to the direction of action the control system takes in response to any measured change in the process variable. If something happens to drive the process variable up, the control system will automatically respond in such a way as to bring the process variable back down where it belongs. If the process variable happens to sag below setpoint, the control system will automatically act to drive the process variable back up to setpoint. Whatever the process variable does in relation to setpoint, the control system takes the opposite (inverse, or negative) action in an attempt to stabilize it at setpoint.

For example, if the unheated process fluid flow rate were to suddenly increase, the heat exchanger outlet temperature would fall due to the physics of heat transfer, but once this drop was detected by the transmitter and reported to the controller, the controller would automatically call for additional steam flow to compensate for the temperature drop, thus bringing the process variable back in agreement with the setpoint. Ideally, a well-designed and well-tuned control loop will sense and

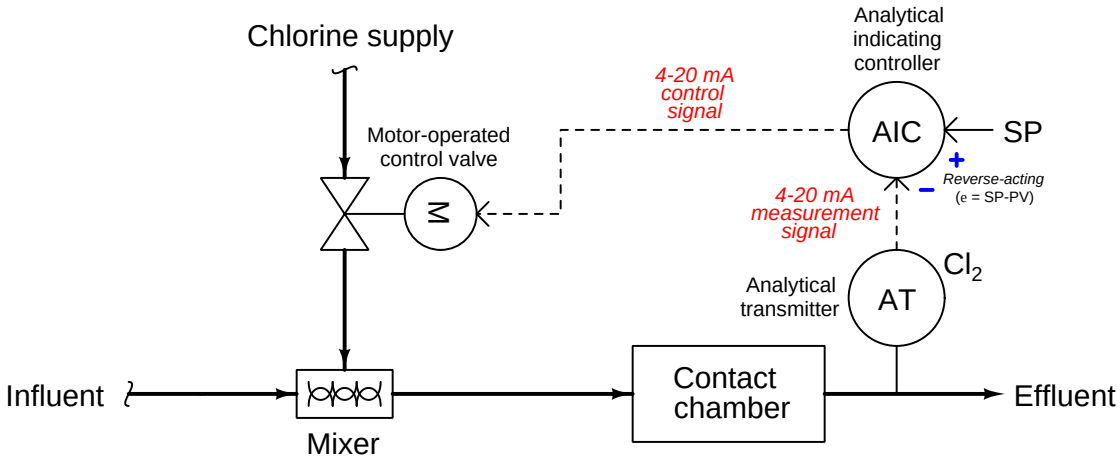
compensate for *any* change in the process or in the setpoint, the end result being a process variable value that always holds steady at the setpoint value.

The unheated fluid flow rate is an example of an uncontrolled, or *wild*, variable because our control system here has no ability to influence it. This flow is also referred to as a *load* because it "loads" or affects the process variable we are trying to stabilize. Loads are present in nearly every controlled system, and indeed are the primary factor necessitating a control system at all. Referring back to our heat exchanger process again, we could adequately control the operating temperature of it with just a manually-set steam control valve if only none of the other factors (steam temperature, fluid flow rate, incoming fluid temperature, etc.) ever changed!

Many types of processes lend themselves to feedback control. Consider an aircraft autopilot system, keeping an airplane on a steady course heading despite the effects of loads such as side-winds: reading the plane's heading (process variable) from an electronic compass and using the rudder as a final control element to change the plane's "yaw." An automobile's "cruise control" is another example of a feedback control system, with the process variable being the car's velocity, and the final control element being the engine's throttle. The purpose of a cruise control is to maintain constant driving speed despite the influence of loads such as hills, head-winds, tail-winds, and road roughness. Steam boilers with automatic pressure controls, electrical generators with automatic voltage and frequency controls, and water pumping systems with automatic flow controls are further examples of how feedback may be used to maintain control over certain process variables.

Modern technology makes it possible to control nearly anything that may be measured in an industrial process. This extends beyond the pale of simple pressure, level, temperature, and flow variables to include even certain chemical properties.

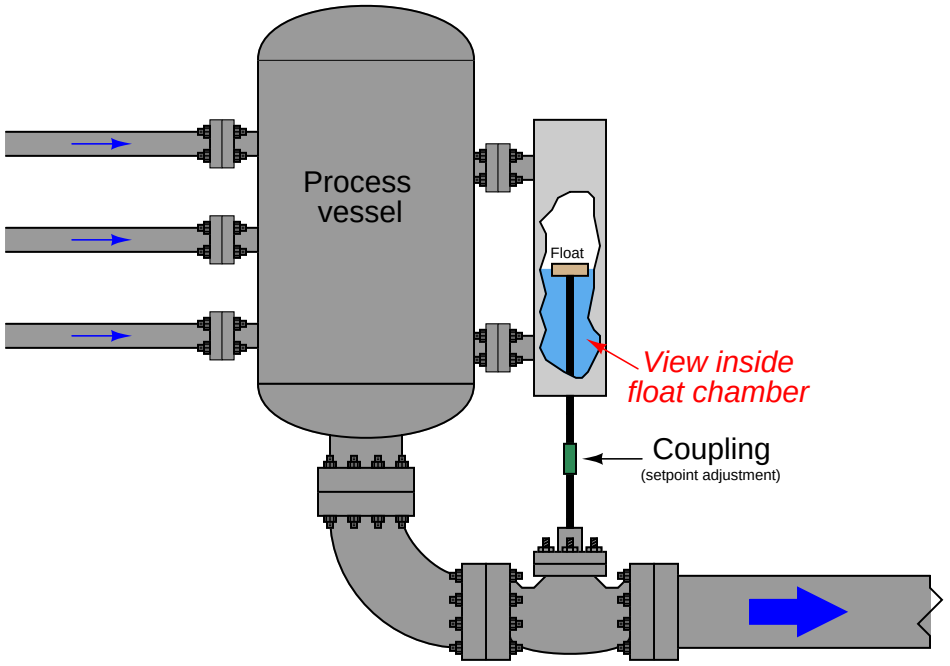
In municipal water and wastewater treatment systems, for example, numerous chemical quantities must be measured and controlled automatically to ensure maximum health and minimum environmental impact. Take for instance the chlorination of treated wastewater, before it leaves the wastewater treatment facility into a large body of water such as a river, bay, or ocean. Chlorine is added to the water to kill any residual bacteria so they do not consume oxygen in the body of water they are released to. Too little chlorine added, and not enough bacteria are killed, resulting in a high *biological oxygen demand* or *BOD* in the water which will asphyxiate the fish swimming in it. Too much chlorine added, and the chlorine itself poses a hazard to marine life. Thus, the chlorine content must be carefully controlled at a particular setpoint, and the control system must take aggressive action if the dissolved chlorine concentration strays too low or too high:



Now that we have seen the basic elements of a feedback control system, we will concentrate on the *algorithms* used in the controller to maintain a process variable at setpoint. For the scope of this topic, an "algorithm" is a mathematical relationship between the process variable and setpoint inputs of a controller, and the output (manipulated variable). Control algorithms determine *how* the manipulated variable quantity is deduced from PV and SP inputs, and range from the elementary to the very complex. In the most common form of control algorithm, the so-called "PID" algorithm, calculus is used to determine the proper final control element action for any combination of input signals.

2.3 Proportional-only control

Imagine a liquid-level control system for a vessel, where the position of a level-sensing float directly sets the stem position of a control valve. As the liquid level rises, the valve opens up proportionally:



Despite its crude mechanical nature, this *proportional* control system would in fact help regulate the level of liquid inside the process vessel. If an operator wished to change the "setpoint" value of this level control system, he or she would have to adjust the coupling between the float and valve stems for more or less distance between the two. Increasing this distance (lengthening the connection) would effectively raise the level setpoint, while decreasing this distance (shortening the connection) would lower the setpoint.

We may generalize the proportional action of this mechanism to describe *any* form of controller where the output is a direct function of process variable (PV) and setpoint (SP):

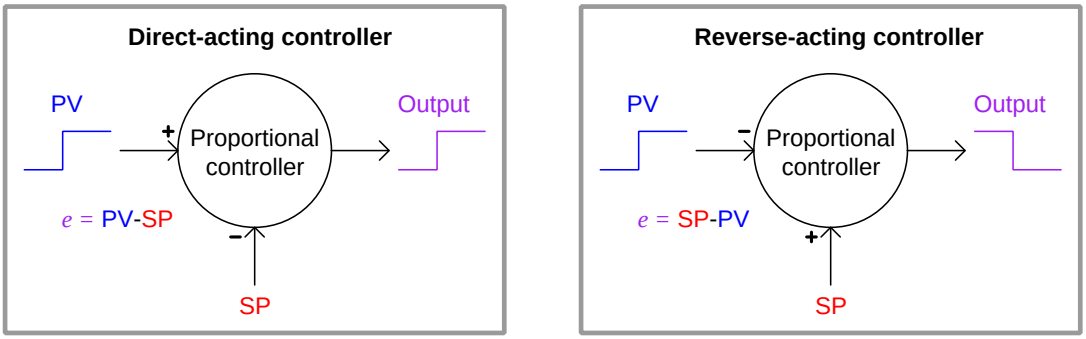
$$m = K_p e + b$$

Where,
m = Controller output
e = Error (difference between PV and SP)
K_p = Proportional gain
b = Bias

A new term introduced with this formula is e, the "error" or difference between process variable and setpoint. Error may be calculated as SP−PV or as PV−SP, depending on whether or not the controller must produce an *increasing* output signal in response to an increase in the process variable ("direct" acting), or output a *decreasing* signal in response to an increase in the process variable ("reverse" acting):

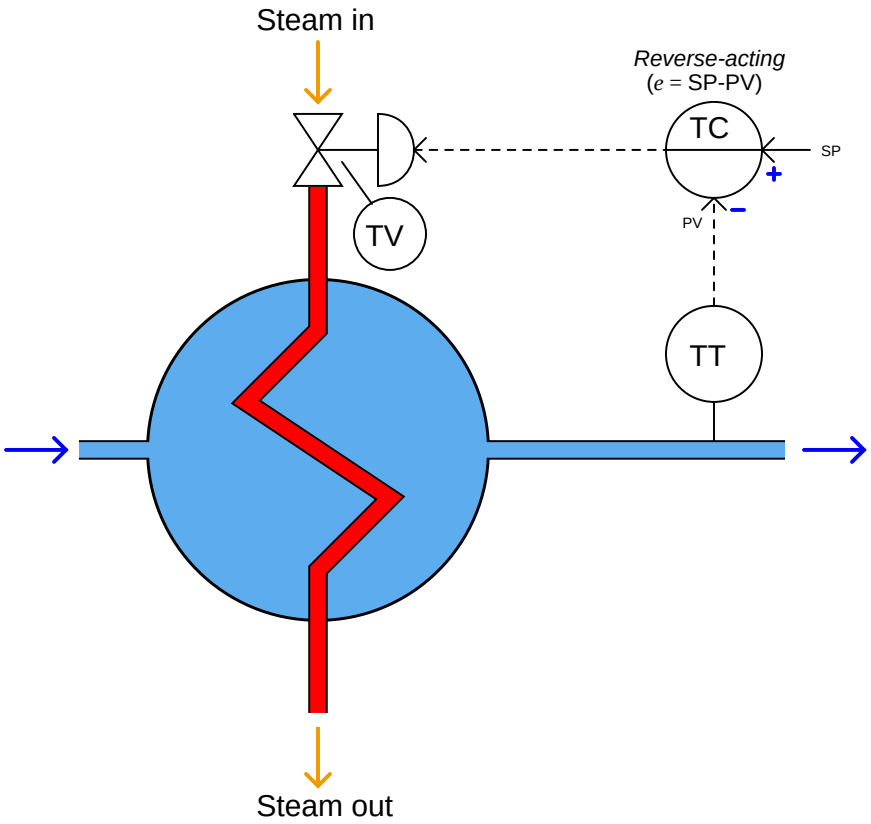
$$m = K_p(\text{PV} - \text{SP}) + b \qquad \text{(Direct-acting proportional controller)}$$

$$m = K_p(\text{SP} - \text{PV}) + b \qquad \text{(Reverse-acting proportional controller)}$$



The optional "+" and "−" symbols clarify the effect each input has on the controller output: a "−" symbol representing an *inverting* effect and a "+" symbol representing a *noninverting* effect. When we say that a controller is "direct-acting" or "reverse-acting" we are referring to it reaction to the PV signal, therefore the output signal from a "direct-acting" controller goes in the same direction as the PV signal and the output from a "reverse-acting" controller goes in the opposite direction of its PV signal. It is important to note, however, that the response to a change in setpoint (SP) will yield the *opposite* response as does a change in process variable (PV): a rising SP will drive the output of a direct-acting controller *down* while a rising SP drives the output of a reverse-acting controller *up*. "+" and "−" symbols explicitly show the effect both inputs have on the controller output, helping to avoid confusion when analyzing the effects of PV changes versus the effects of SP changes.

The direction of action required of the controller is determined by the nature of the process, transmitter, and final control element. In the case of the crude mechanical level controller, the action needs to be *direct* so that a greater liquid level will result in a further-open control valve to drain the vessel faster. In the case of the automated heat exchanger shown earlier, we are assuming that an increasing output signal sent to the control valve results in increased steam flow, and consequently higher temperature, so our controller will need to be reverse-acting (i.e. an increase in measured temperature results in a decrease in output signal; error calculated as SP−PV):



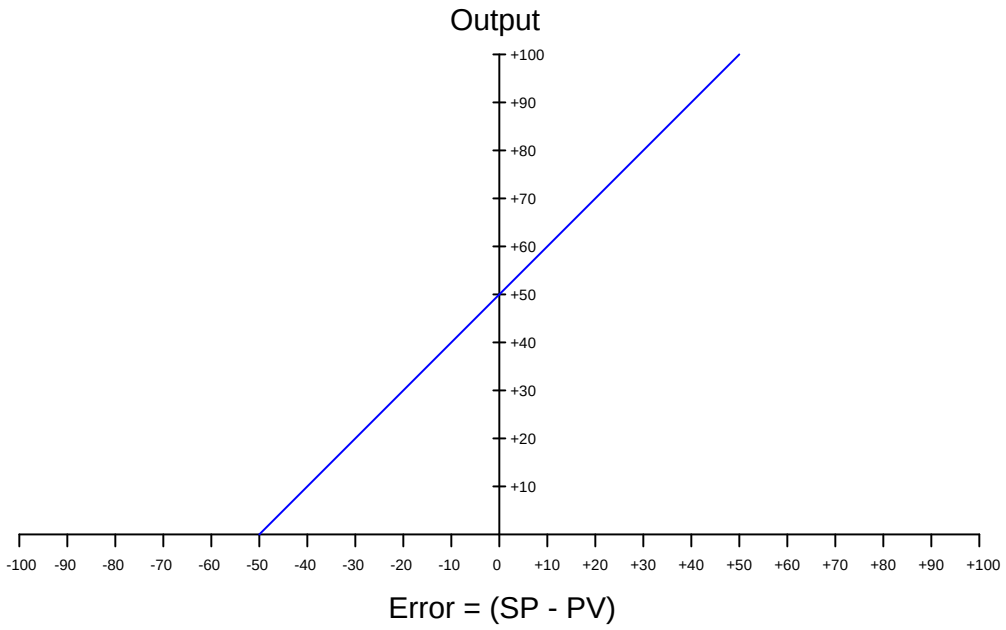
After the error has been calculated, the controller then multiplies the error signal by a constant value called the *gain*, which is programmed into the controller. The resulting figure, plus a "bias" quantity, becomes the output signal sent to the valve to proportion it. The "gain" value is exactly what it seems to be for anyone familiar with electronic amplifier circuits: a ratio of output to input. In this case, the gain of a proportional controller is the ratio of output signal change to input signal change, or how *aggressive* the controller reacts to changes in input (PV or SP).

To give a numerical example, a loop controller set to have a gain of 4 will change its output signal by 40% if it sees an input change of 10%: the ratio of output change to input change will be 4:1. Whether the input change comes in the form of a setpoint adjustment, a drift in the process variable, or some combination of the two does not matter to the magnitude of the output change.

The bias value of a proportional controller is simply the value of its output whenever process

variable happens to be equal to setpoint (i.e. a condition of zero *error*). Without a bias term in the proportional control formula, the valve would always return to a fully shut (0%) condition if ever the process variable reached the setpoint value. The bias term allows the final control element to achieve a non-zero state at setpoint.

If the $m = K_p e + b$ proportional controller formula resembles the standard slope-intercept form of linear equation ($y = mx + b$), it is more than coincidence. Often, the response of a proportional controller is shown graphically as a line, the slope of the line representing gain and the y-intercept of the line representing the output bias point, or what value the output signal will be when there is no error (PV precisely equals SP):

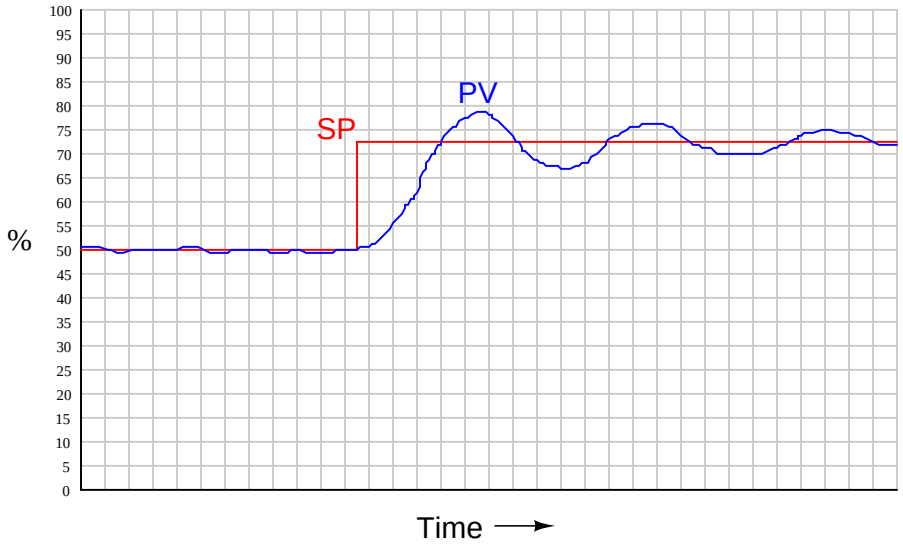


In this graph the bias value is 50% and the gain of the controller is 1. Changing the bias value (b) of the controller shifts the line up or down. Changing the gain value (K_p) alters the slope of the line for more or less aggressive control action.

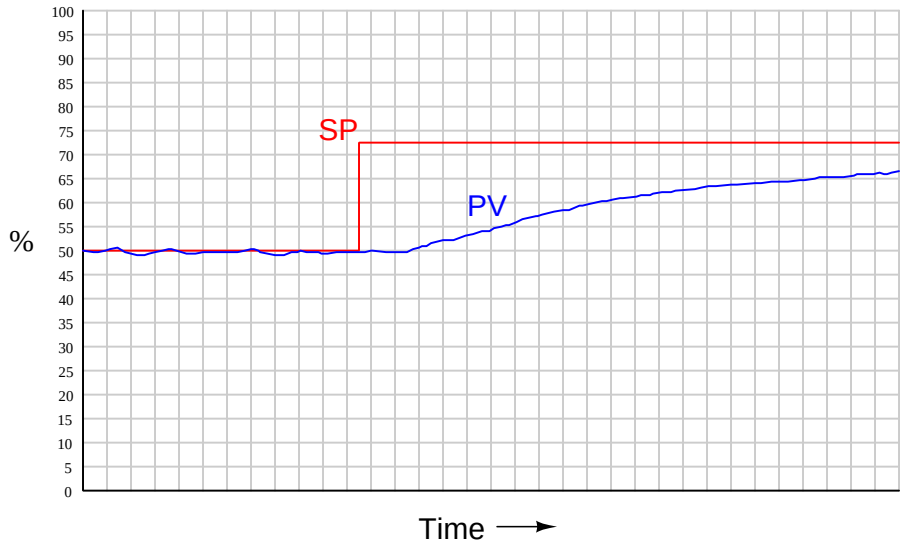
If the controller could be configured for infinite gain, its response would duplicate on/off control. That is, *any* amount of error will result in the output signal becoming "saturated" at either 0% or 100%, and the final control element will simply turn on fully when the process variable drops below setpoint and turn off fully when the process variable rises above setpoint. Conversely, if the controller is set for zero gain, it will become completely unresponsive to changes in either process variable *or* setpoint: the valve will hold its position at the bias point no matter what happens to the process.

Obviously, then, we must set the gain somewhere between infinity and zero in order for this algorithm to function any better than on/off control. Just how much gain a controller needs to have depends on the process and all the other instruments in the control loop.

If the gain is set too high, there will be oscillations as the PV converges on a new setpoint value:

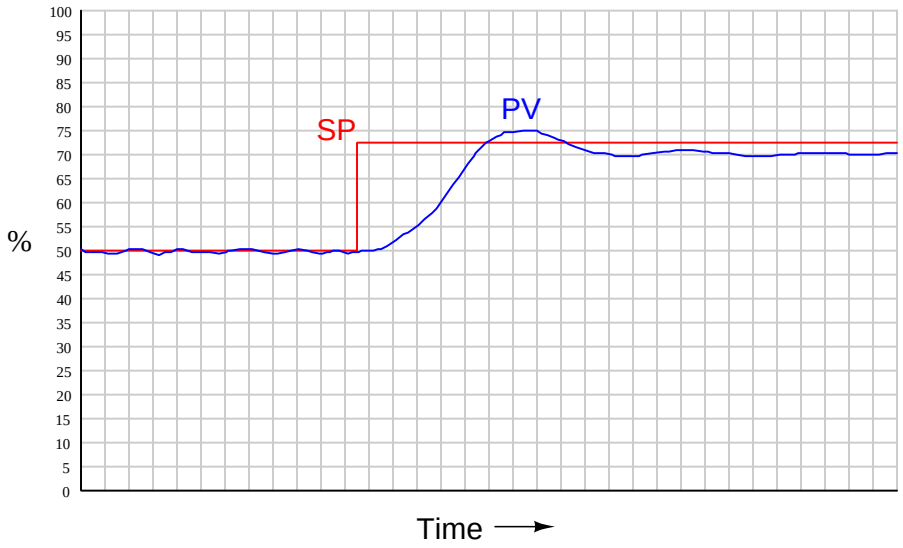

100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
%
SP
PV
Time

If the gain is set too low, the process response will be stable under steady-state conditions but relatively slow to respond to changes in setpoint, as shown in the following trend recording:



A characteristic deficiency of proportional control action, exacerbated with low controller gain values, is a phenomenon known as *proportional-only offset* where the PV never fully reaches SP. A full explanation of proportional-only offset is too lengthy for this discussion and will be presented in a subsequent section of the book, but may be summarized here simply by drawing attention to the proportional controller equation which tells us the output always returns to the bias value when PV reaches SP (i.e. $m = b$ when PV = SP). If anything changes in the process to require a different output value than the bias (b) to stabilize the PV, an error between PV and SP *must* develop to drive the controller output to that necessary output value. This means it is only by chance that the PV will settle precisely at the SP value – most of the time, the PV will deviate from SP in order to generate an output value sufficient to stabilize the PV and prevent it from drifting. This persistent error, or offset, worsens as the controller gain is reduced. Increasing controller gain causes this offset to decrease, but at the expense of oscillations.

With proportional-only control, the choice of gain values is really a compromise between excessive oscillations and excessive offset. A well-tuned proportional controller response is shown here:



An unnecessarily confusing aspect of proportional control is the existence of two completely different ways to express controller proportionality. In the proportional-only equation shown earlier, the degree of proportional action was specified by the constant K_p, called *gain*. However, there is another way to express the sensitivity of proportional action, and that is to state the percentage of error change necessary to make the output (m) change by 100%. Mathematically, this is the inverse of gain, and it is called *proportional band* (PB):

$$K_p = \frac{1}{\text{PB}} \qquad \text{PB} = \frac{1}{K_p}$$

Gain is always specified as a unitless value[1], whereas proportional band is always specified as a percentage. For example, a gain value of 2.5 is equivalent to a proportional band value of 40%, because the error input to this controller must change by 40% in order to make the output change a full 100%.

[1] In electronics, the unit of *decibels* is commonly used to express gains. Thankfully, the world of process control was spared the introduction of decibels as a unit of measurement for controller gain. The last thing we need is a *third* way to express the degree of proportional action in a controller!

Due to the existence of these two completely opposite conventions for specifying proportional action, you may see the proportional term of the control equation written differently depending on whether the author assumes the use of gain or the use of proportional band:

$$K_p = \text{gain} \qquad \text{PB} = \text{proportional band}$$

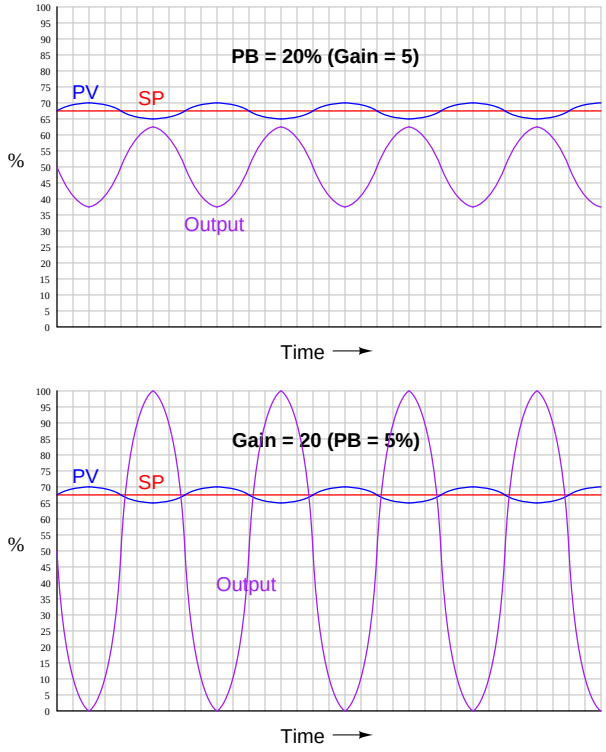
$$K_p e \qquad\qquad \frac{1}{\text{PB}} e$$

Many modern digital electronic controllers allow the user to conveniently select the unit they wish to use for proportional action. However, even with this ability, anyone tasked with adjusting a controller's "tuning" values may be required to translate between gain and proportional band, especially if certain values are documented in a way that does not match the unit configured for the controller.

When you communicate the proportional action setting of a process controller, you should always be careful to specify either "gain" or "proportional band" to avoid ambiguity. *Never* simply say something like, "The proportional setting is twenty," for this could mean either:

- Proportional band = 20%; Gain = 5 . . . *or* . . .
- Gain = 20; Proportional band = 5%

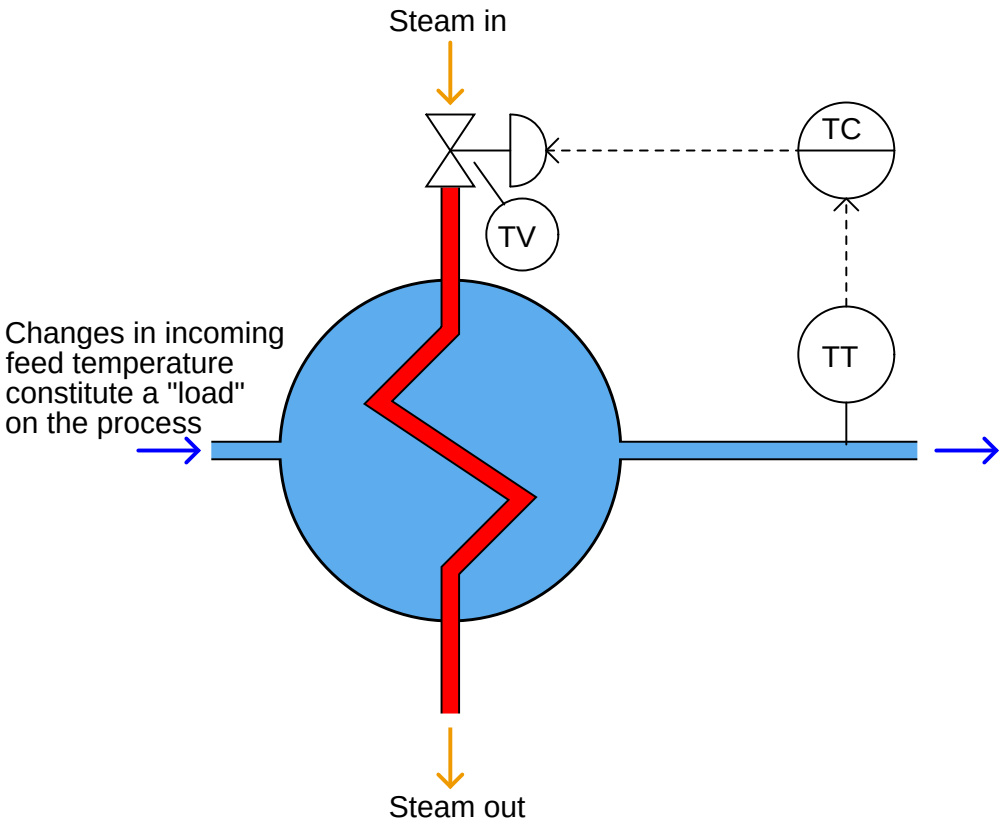
As you can see here, the real-life difference in controller response to an input disturbance (wave) depending on whether it has a proportional band of 20% or a gain of 20 is quite dramatic:



2.4 Proportional-only offset

A fundamental limitation of proportional control has to do with its response to changes in setpoint and changes in process *load*. A "load" in a controlled process is any variable not controlled by the loop controller which nevertheless affects the process variable the controller is trying to regulate. In other words, a "load" is any factor the loop controller must compensate for while maintaining the process variable at setpoint.

In our hypothetical heat exchanger system, the temperature of the incoming process fluid is an example of a load:



If the incoming fluid temperature were to suddenly decrease, the immediate effect this would have on the process would be to decrease the outlet temperature (which is the temperature we are trying to maintain at a steady value). It should make intuitive sense that a colder incoming fluid will require more heat input to raise it to the same outlet temperature as before. If the heat input remains the same (at least in the immediate future), this colder incoming flow must make the outlet flow colder than it was before. Thus, incoming feed temperature has an impact on the outlet temperature whether we like it or not, and the control system must compensate for these unforeseen and uncontrolled changes. This is precisely the definition of a "load": a burden[2] on the control

[2] One could argue that the presence of loads actually *justifies* a control system, for if there were no loads, there

system.

Of course, it is the job of the controller to counteract any tendency for the outlet temperature to stray from setpoint, but as we shall soon see this cannot be perfectly achieved with proportional control alone.

Let us perform a "thought experiment" to demonstrate this phenomenon of proportional-only offset. Imagine the controller has been controlling outlet temperature exactly at setpoint (PV = SP), and then suddenly the inlet feed temperature drops and remains colder than before. Recall that the equation for a reverse-acting proportional controller is as follows:

$$m = K_p(\text{SP} - \text{PV}) + b$$

Where,
m = Controller output
K_p = Proportional gain
SP = Setpoint
PV = Process variable
b = Bias

The introduction of colder feed fluid to the heat exchanger makes the outlet temperature (PV) begin to fall. As the PV falls, the controller calculates a positive error (SP − PV). This positive error, when multiplied by the controller's gain value, drives the output to a greater value. This opens up the steam valve, adding more heat to the exchanger.

As more heat is added, the rate of temperature drop slows down. The further the PV drops, the more the steam valve opens, until enough additional heat is being added to the heat exchanger to maintain a constant outlet temperature. However, this new stable PV value will be less than it was prior to the introduction of colder feed (i.e. less than the SP). In fact, the controller's automatic action can *never* return the PV to its original (SP) value so long as the feed remains colder than before. The reason for this is that a greater flow of steam is necessary to balance a colder feed coming in, and the only way a proportional controller is ever going to automatically drive the steam valve to this greater-flow position is if an error develops between PV and SP. Thus, an *offset* inevitably develops between PV and SP due to the load (colder feed).

We may prove the inevitability of this offset another way: imagine somehow that the PV did actually return to the SP value despite the colder feed fluid (remaining colder). If this happened, the steam valve would also return to its former throttling position where it was before the feed temperature dropped. However, we know that this former position will not allow enough steam through to the exchanger to overcome the colder feed – if it did, the PV never would have decreased to begin with! A further-open valve is precisely what we need to stabilize the PV given this colder feed, yet the only way the proportional-only controller can achieve this is if the PV actually falls below SP.

To summarize: the only way a proportional-only controller can automatically generate a new output value (m) is if the PV deviates from SP. Therefore, load changes (requiring new output values to compensate) force the PV to deviate from SP.

---

would be nothing to compensate for, and therefore no need for an automatic control system at all! In the total absence of loads, a manually-set final control element would be enough to hold most process variables at setpoint.

Another "thought experiment" may be helpful to illustrate the phenomenon of proportional-only offset. Imagine building your own cruise control system for your automobile based on the proportional-only equation: the engine's throttle position is a function of the difference between PV (road speed) and SP (the desired "target" speed). Let us further suppose that you carefully adjust the bias value of your cruise control system to achieve PV = SP on level ground at a speed of 70 miles per hour (70% on a 0 to 100 MPH speedometer scale), with the throttle at a position of 40%, and a gain (K_p) of 2:

$$m = K_p(\text{SP} - \text{PV}) + b$$

$$40\% = 2(70 - 70) + 40\%$$

Imagine now that after cruising precisely at setpoint (70% = 70 MPH), the road begins to incline uphill for several miles. This, obviously, is a load on the cruise control system. With the cruise control disengaged, the automobile would slow down because the same throttle position (40%) sufficient to maintain setpoint (70 MPH) on level ground is not enough power to maintain that same setpoint on an incline.

With the cruise control engaged, the engine throttle will automatically open further as speed drops. At a speed of 69 MPH, the throttle opens up to 42%. At a speed of 68 MPH, the throttle opens up to 44%. Every drop in speed of 1 MPH results in a 2% further-open throttle to send more power to the wheels.

Suppose the demands of this particular inclined road require a 50% throttle position for this automobile to maintain a constant speed. In order for your proportional-only cruise control system to deliver this necessary 50% throttle position, the speed will have to "droop" by 5 MPH below setpoint:

$$m = K_p(\text{SP} - \text{PV}) + b$$

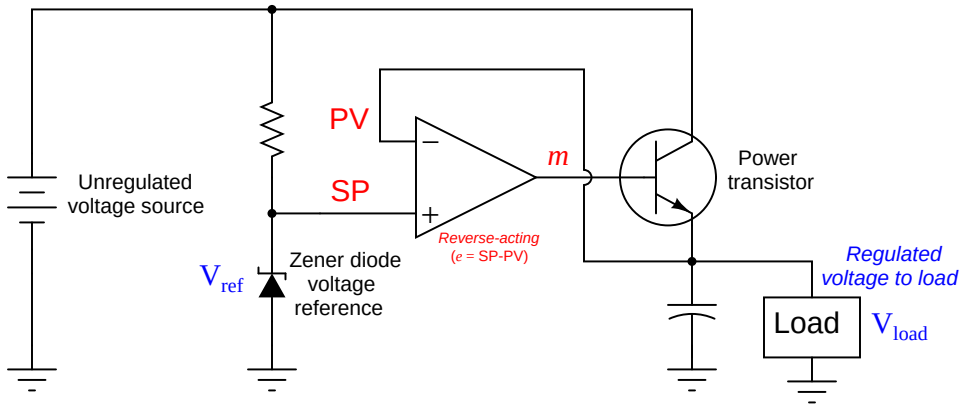
$$50\% = 2(70 - 65) + 40\%$$

There is simply no other way for your proportional-only controller to automatically achieve the requisite 50% throttle position aside from letting the speed sag below setpoint by 5% (5 MPH). Given this fact, the only way the proportional-only cruise control will ever return the speed to setpoint (70 MPH) is if and when the load conditions change to allow for a lesser throttle position of 40%. So long as the load demands a different throttle position than the bias value, the speed *must* deviate from the setpoint value of 70 MPH.

This necessary error developing between PV and SP is called *proportional-only offset*, sometimes called *droop*. The amount of droop depends on how severe the load change is, and how aggressive the controller responds (i.e. how much gain it has). The term "droop" is very misleading, as it is possible for the error to develop the other way (i.e. the PV might rise above SP due to a load change!). Imagine the opposite load-change scenario in our steam heat exchanger process, where the incoming feed temperature suddenly *rises* instead of falls. If the controller was controlling exactly at setpoint before this upset, the final result will be an outlet temperature that settles at some point *above* setpoint, enough so the controller is able to pinch the steam valve far enough closed to stop any further rise in temperature.

Proportional-only offset also occurs as a result of setpoint changes. We could easily imagine the same sort of effect following an operator's increase of setpoint for the temperature controller on the heat exchanger. After increasing the setpoint, the controller immediately increases the output signal, sending more steam to the heat exchanger. As temperature rises, though, the proportional algorithm causes the output signal to decrease. When the rate of heat energy input by the steam equals the rate of heat energy carried away from the heat exchanger by the heated fluid (a condition of *energy balance*), the temperature stops rising. This new equilibrium temperature will not be at setpoint, assuming the temperature was holding at setpoint prior to the human operator's setpoint increase. The new equilibrium temperature indeed *cannot* ever achieve any setpoint value higher than the one it did in the past, for if the error ever returned to zero (PV = SP), the steam valve would return to its old position, which we know would be insufficient to raise the temperature of the heated fluid to a new value.

An example of proportional-only control in the context of electronic power supply circuits is the following opamp voltage regulator, used to stabilize voltage to a load with power supplied by an unregulated voltage source:



In this circuit, a zener diode establishes a "reference" voltage (which may be thought of as a "setpoint" for the controlling opamp to follow). The operational amplifier acts as the proportional-only controller, sensing voltage at the load (PV), and sending a driving output voltage to the base of the power transistor to keep load voltage constant despite changes in the supply voltage or changes in load current (both "loads" in the process-control sense of the word, since they tend to influence voltage at the load circuit without being under the control of the opamp).

If everything functions properly in this voltage regulator circuit, the load's voltage will be stable over a wide range of supply voltages and load currents. However, the load voltage cannot ever *precisely* equal the reference voltage established by the zener diode, even if the operational amplifier (the "controller") is without defect. The reason for this incapacity to perfectly maintain "setpoint" is the simple fact that in order for the opamp to generate any output signal at all, there *absolutely must be* a differential voltage between the two input terminals for the amplifier to amplify. Operational amplifiers (ideally) generate an output voltage equal to the enormously high gain value (A_V) multiplied by the difference in input voltages (in this case, $V_{ref} - V_{load}$). If V_{load} (the "process

variable") were to ever achieve equality with V_{ref} (the "setpoint"), the operational amplifier would experience absolutely no differential input voltage to amplify, and its output signal driving the power transistor would fall to zero. Therefore, there must always exist some *offset* between V_{load} and V_{ref} (between process variable and setpoint) in order to give the amplifier some input voltage to amplify.

The amount of offset is ridiculously small in such a circuit, owing to the enormous gain of the operational amplifier. If we take the opamp's transfer function to be $V_{out} = A_V(V_{(+)} - V_{(-)})$, then we may set up an equation predicting the load voltage as a function of reference voltage (assuming a constant 0.7 volt drop between the base and emitter terminals of the transistor):

$$V_{out} = A_V(V_{(+)} - V_{(-)})$$

$$V_{out} = A_V(V_{ref} - V_{load})$$

$$V_{load} + 0.7 = A_V(V_{ref} - V_{load})$$

$$V_{load} + 0.7 = A_V V_{ref} - A_V V_{load}$$

$$V_{load} + A_V V_{load} = A_V V_{ref} - 0.7$$

$$(A_V + 1)V_{load} = A_V V_{ref} - 0.7$$

$$V_{load} = \frac{A_V V_{ref} - 0.7}{A_V + 1}$$

If, for example, our zener diode produced a reference voltage of 5.00000 volts and the operational amplifier had an open-loop voltage gain of 250000, the load voltage would settle at a theoretical value of 4.9999772 volts: just barely below the reference voltage value. If the opamp's open-loop voltage gain were much less – say only 100 – the load voltage would only be 4.94356 volts. This still is quite close to the reference voltage, but definitely not as close as it would be with a greater opamp gain!

Clearly, then, we can minimize proportional-only offset by increasing the gain of the process controller gain (i.e. decreasing its proportional band). This makes the controller more "aggressive" so it will move the control valve further for any given change in PV or SP. Thus, not as much error needs to develop between PV and SP to move the valve to any new position it needs to go. However, too much controller gain makes the control system unstable: at best it will exhibit residual oscillations after setpoint and load changes, and at worst it will oscillate out of control altogether. Extremely high gains work well to minimize offset in operational amplifier circuits, only because time delays are negligible between output and input. In applications where large physical processes are being controlled (e.g. furnace temperatures, tank levels, gas pressures, etc.) rather than voltages across small electronic loads, such high controller gains would be met with debilitating oscillations.

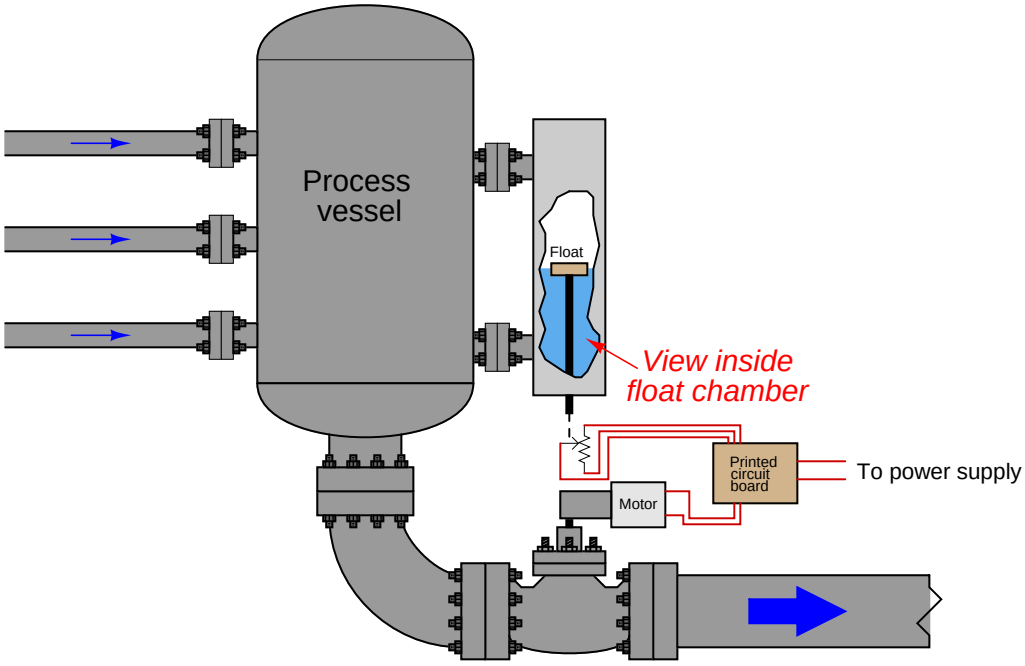
If we are limited in how much gain we can program in to the controller, how do we minimize this offset? One way is for a human operator to periodically place the controller in manual mode and move the control valve just a little bit more so the PV once again reaches SP, then place the controller back

into automatic mode. In essence this technique adjusts the "Bias" term of the controller equation. The disadvantage of this technique is rather obvious: it requires human intervention. What is the point of having an automation system requiring periodic human intervention to maintain setpoint?

A more sophisticated method for eliminating proportional-only offset is to add a different control action to the controller: one that takes action based on the amount of error between PV and SP and the amount of time that error has existed. We call this control mode *integral*, or *reset*.

2.5 Integral (reset) control

Imagine a liquid-level control system for a vessel, where the position of a level-sensing float sets the position of a potentiometer, which then sets the *speed* of a motor-actuated control valve. If the liquid level is above setpoint, the valve continually opens up; if below setpoint, the valve continually closes off:



Unlike the *proportional* control system where valve position was a direct function of float position, this control system sets the *speed* of the motor-driven valve according to the float position. The further away from setpoint the liquid level is, the *faster* the valve moves open or closed. In fact, the only time the valve will ever halt its motion is when the liquid level is precisely at setpoint; otherwise, the control valve will be in constant motion.

This control system does its job in a very different manner than the all-mechanical float-based proportional control system illustrated previously. Both systems are capable of regulating liquid level inside the vessel, but they take very different approaches to doing so. One of the most significant differences in control behavior is how the proportional system would inevitably suffer from *offset* (a persistent error between PV and SP), whereas this control system actively works at all times to eliminate offset. The motor-driven control valve literally does not rest until all error has been eliminated!

Instead of characterizing this control system as *proportional*, we call it *integral*[3] in honor of the calculus principle ("integration") whereby small quantities are accumulated over some span to form a total. Don't let the word "calculus" scare you! You are probably already familiar with the concept of numerical integration even though you may have never heard of the term before.

Calculus is a form of mathematics dealing with *changing* variables, and how rates of change relate between different variables. When we "integrate" a variable with respect to time, what we are doing is *accumulating* that variable's value as time progresses. Perhaps the simplest example of this is a vehicle odometer, accumulating the total distance traveled by the vehicle over a certain time period. This stands in contrast to a speedometer, indicating the rate of distance traveled *per* unit of time.

Imagine a car moving along at exactly 30 miles per hour. How far will this vehicle travel after 1 hour of driving this speed? Obviously, it will travel 30 miles. Now, how far will this vehicle travel if it continues for another 2 hours at the exact same speed? Obviously, it will travel 60 more miles, for a total distance of 90 miles since it began moving. If the car's speed is a constant, calculating total distance traveled is a simple matter of multiplying that speed by the travel time.

The odometer mechanism that keeps track of the mileage traveled by the car may be thought of as *integrating* the speed of the car with respect to time. In essence, it is multiplying speed times time continuously to keep a running total of how far the car has gone. When the car is traveling at a high speed, the odometer "integrates" at a faster rate. When the car is traveling slowly, the odometer "integrates" slowly.

If the car travels in reverse, the odometer will decrement (count down) rather than increment (count up) because it sees a negative quantity for speed[4]. The rate at which the odometer decrements depends on how fast the car travels in reverse. When the car is stopped (zero speed), the odometer holds its reading and neither increments nor decrements.

Now let us return to the context of an automated process to see how this calculus principle works inside a process controller. Integration is provided either by a pneumatic mechanism, an electronic opamp circuit, or by a microprocessor executing a digital integration algorithm. The variable being integrated is *error* (the difference between PV and SP) over time. Thus the integral mode of the controller ramps the output either up or down over time in response to the amount of error existing between PV and SP, and the sign of that error. We saw this "ramping" action in the behavior of the liquid level control system using a motor-driven control valve commanded by a float-positioned potentiometer: the valve stem continuously moves so long as the liquid level deviates from setpoint. The reason for this ramping action is to increase or decrease the output *as far as it is necessary* in order to completely eliminate any error and force the process variable to precisely equal setpoint. Unlike proportional action, which simply moves the output an amount proportional to any change in PV or SP, integral control action never stops moving the output until all error is eliminated.

[3] An older term for this mode of control is *floating*, which I happen to think is particularly descriptive. With a "floating" controller, the final control element continually "floats" to whatever value it must in order to completely eliminate offset.

[4] At least the old-fashioned mechanical odometers would. Modern cars use a pulse detector on the driveshaft which cannot tell the difference between forward and reverse, and therefore their odometers always increment. Shades of the movie *Ferris Bueller's Day Off*.

If proportional action is defined by the error telling the output how *far* to move, integral action is defined by the error telling the output how *fast* to move. One might think of integral as being how "impatient" the controller is, with integral action constantly ramping the output as far as it needs to go in order to eliminate error. Once the error is zero (PV = SP), of course, the integral action stops ramping, leaving the controller output (valve position) at its last value just like a stopped car's odometer holds a constant value.

If we add an integral term to the controller equation, we get something that looks like this[5]:

$$m = K_p e + \frac{1}{\tau_i} \int e \, dt + b$$

Where,

m = Controller output
e = Error (difference between PV and SP)
K_p = Proportional gain
τ_i = Integral time constant (minutes)
t = Time
b = Bias

The most confusing portion of this equation for those new to calculus is the part that says "$\int e \, dt$". The integration symbol (looks like an elongated letter "S") tells us the controller will accumulate ("sum") multiple products of error (e) over tiny slices of time (dt). Quite literally, the controller multiplies error by time (for very short segments of time, dt) and continuously adds up all those products to contribute to the output signal which then drives the control valve (or other final control element). The integral time constant (τ_i) is a value set by the technician or engineer configuring the controller, proportioning this cumulative action to make it more or less aggressive over time.

To see how this works in a practical sense, let's imagine how a proportional + integral controller would respond to the scenario of a heat exchanger whose inlet temperature suddenly dropped. As we saw with proportional-only control, an inevitable offset occurs between PV and SP with changes in load, because an error *must* develop if the controller is to generate the different output signal value necessary to halt further change in PV. We called this effect *proportional-only offset*.

Once this error develops, though, integral action begins to work. Over time, a larger and larger quantity accumulates in the integral mechanism (or register) of the controller due to the persistent error between PV and SP. That accumulated value adds to the controller's output, driving the steam control valve further and further open. This, of course, adds heat at a faster rate to the heat exchanger, which causes the outlet temperature to rise. As the temperature re-approaches setpoint, the error becomes smaller and thus the integral action proceeds at a slower rate (like a car's odometer incrementing at a slower rate as the car's speed decreases). So long as the PV is below SP (the outlet temperature is still too cool), the controller will continue to integrate upwards, driving the control valve further and further open. Only when the PV rises to exactly meet SP does

[5]The equation for a proportional + integral controller is often written without the bias term (b), because the presence of integral action makes it unnecessary. In fact, if we let the integral term completely replace the bias term, we may consider the integral term to be a self-*resetting* bias. This, in fact, is the meaning of the word "reset" in the context of PID controller action: the "reset" term of the controller acts to eliminate offset by continuously adjusting (resetting) the bias as necessary.

integral action finally rest, holding the valve at a steady position. Integral action tirelessly works to eliminate any offset between PV and SP, thus neatly eliminating the offset problem experienced with proportional-only control action.

As with proportional action, there are (unfortunately) two completely opposite ways to specify the degree of integral action offered by a controller. One way is to specify integral action in terms of *minutes* or *minutes per repeat*. A large value of "minutes" for a controller's integral action means a less aggressive integral action over time, just as a large value for proportional band means a less aggressive proportional action. The other way to specify integral action is the inverse: how many *repeats per minute*, equivalent to specifying proportional action in terms of gain (large value means aggressive action). For this reason, you will sometimes see the integral term of a PID equation written differently:

$$\tau_i = \text{minutes per repeat} \qquad K_i = \text{repeats per minute}$$

$$\frac{1}{\tau_i} \int e\, dt \qquad K_i \int e\, dt$$

Many modern digital electronic controllers allow the user to select the unit they wish to use for integral action, just as they allow a choice between specifying proportional action as gain or as proportional band.

Integral is a highly effective mode of process control. In fact, some processes respond so well to integral controller action that it is possible to operate the control loop on integral action alone, without proportional. Typically, though, process controllers implement some form of proportional plus integral ("PI") control.

Just as too much proportional gain will cause a process control system to oscillate, too much integral action (i.e. an integral time constant that is too short) will also cause oscillation. If the integration happens at too fast a rate, the controller's output will "saturate" either high or low before the process variable can make it back to setpoint. Once this happens, the only condition that will "unwind" the accumulated integral quantity is for an error to develop of the opposite sign, and remain that way long enough for a canceling quantity to accumulate. Thus, the PV must cross over the SP, guaranteeing at least another half-cycle of oscillation.

A similar problem called *reset windup* (or *integral windup*) happens when external conditions make it impossible for the controller to achieve setpoint. Imagine what would happen in the heat exchanger system if the steam boiler suddenly stopped producing steam. As outlet temperature dropped, the controller's proportional action would open up the control valve in a futile effort to raise temperature. If and when steam service is restored, proportional action would just move the valve back to its original position as the process variable returned to its original value (before the boiler died). This is how a proportional-only controller would respond to a steam "outage": nice and predictably. If the controller had integral action, however, a much worse condition would result. All the time spent with the outlet temperature below setpoint causes the controller's integral term to "wind up" in a futile attempt to admit more steam to the heat exchanger. This accumulated quantity can only be un-done by the process variable rising above setpoint for an equal error-time product[6], which means when the steam supply resumes, the temperature will rise well above setpoint

[6]Since integration is fundamentally a process of multiplication followed by addition, the units of measurement are always the product (multiplication) of the function's variables. In the case of reset (integral) control, we are multiplying

until the integral action finally "unwinds" and brings the control valve back to a same position again.

Various techniques exist to manage integral windup. Controllers may be built with limits to restrict how far the integral term can accumulate under adverse conditions. In some controllers, integral action may be turned off completely if the error exceeds a certain value. The surest fix for integral windup is human operator intervention, by placing the controller in manual mode. This typically resets the integral accumulator to a value of zero and loads a new value into the bias term of the equation to set the valve position wherever the operator decides. Operators usually wait until the process variable has returned at or near setpoint before releasing the controller into automatic mode again.

While it might appear that operator intervention is again a problem to be avoided (as it was in the case of having to correct for proportional-only offset), it is noteworthy to consider that the conditions leading to integral windup usually occur only during shut-down conditions. It is customary for human operators to run the process manually anyway during a shutdown, and so the switch to manual mode is something they would do anyway and the potential problem of windup often never manifests itself.

Integral control action has the unfortunate tendency to create loop oscillations ("cycling") if the final control element exhibits hysteresis, such as the case with a "sticky" control valve. Imagine for a moment our steam-heated heat exchanger system where the steam control valve possesses excessive packing friction and therefore refuses to move until the applied air pressure changes far enough to overcome that friction, at which point the valve "jumps" to a new position and then "sticks" in that new position. If the valve happens to stick at a stem position resulting in the product temperature settling slightly below setpoint, the controller's integral action will continually increase the output signal going to the valve in an effort to correct this error (as it should). However, when that output signal has risen far enough to overcome valve friction and move the stem further open, it is very likely the stem will once again "stick" but this time do so at a position making the product temperature settle *above* setpoint. The controller's integral action will then ramp downward in an effort to correct this new error, but due to the valve's friction making precise positioning impossible, the controller can never achieve setpoint and therefore it cyclically "hunts" above and below setpoint.

The best solution to this "reset cycling" phenomenon, of course, is to correct the hysteresis in the final control element. Eliminating friction in the control valve will permit precise positioning and allow the controller's integral action to achieve setpoint as designed. Since it is practically impossible to eliminate *all* friction from a control valve, however, other solutions to this problem exist. One of them is to program the controller to stop integrating whenever the error is less than some pre-configured value (sometimes referred to as the "integral deadband" or "reset deadband" of the controller). By activating reset control action only for significant error values, the controller ignores small errors rather than "compulsively" trying to correct for any detected error no matter how small.

---

controller error (the difference between PV and SP, usually expressed in percent) by time (usually expressed in minutes or seconds). Therefore the result will be an "error-time" product. In order for an integral controller to self-recover following windup, the error must switch signs and the error-time product accumulate to a sufficient value to cancel out the error-time product accumulated during the windup period.

2.6 Derivative (rate) control

The final element of PID control is the "D" term, which stands for *derivative*. This is a calculus concept like integral, except most people consider it easier to understand. Simply put, derivative is the expression of a variable's *rate-of-change* with respect to another variable. Finding the derivative of a function (differentiation) is the inverse operation of integration. With integration, we calculated accumulated value of some variable's product with time. With derivative, we calculate the ratio of a variable's change per unit of time. Whereas integration is fundamentally a multiplicative operation (products), differentiation always involves division (ratios).

A controller with derivative (or *rate*) action looks at how fast the process variable changes per unit of time, and takes action proportional to that rate of change. In contrast to integral (reset) action which represents the "impatience" of the controller, derivative (rate) action represents the "caution" of the controller.

If the process variable starts to change at a high rate of speed, the job of derivative action is to move the final control element in such a direction as to counteract this rapid change, and thereby moderate the speed at which the process variable changes. In simple terms, derivative action works to limit how fast the error can change.

What this will do is make the controller "cautious" with regard to rapid changes in process variable. If the process variable is headed toward the setpoint value at a rapid rate, the derivative term of the equation will diminish the output signal, thus tempering the controller's response and slowing the process variable's approach toward setpoint. This is analogous to a truck driver preemptively applying the brakes to slow the approach to an intersection, knowing that the heavy truck doesn't "stop on a dime." The heavier the truck's load, the sooner a cautious driver will apply the brakes, to avoid "overshoot" beyond the stop sign and into the intersection. For this reason, derivative control action is also called *pre-act* in addition to being called *rate*, because it acts "ahead of time" to avoid overshoot.

If we modify the controller equation to incorporate differentiation, it will look something like this:

$$m = K_p e + \frac{1}{\tau_i} \int e \, dt + \tau_d \frac{de}{dt} + b$$

Where,

m = Controller output
e = Error (difference between PV and SP)
K_p = Proportional gain
τ_i = Integral time constant (minutes)
τ_d = Derivative time constant (minutes)
t = Time
b = Bias

The $\frac{de}{dt}$ term of the equation expresses the rate of change of error (e) over time (t). The lower-case letter "d" symbols represent the calculus concept of *differentials* which may be thought of in this context as very tiny increments of the following variables. In other words, $\frac{de}{dt}$ refers to the ratio of a very small change in error (de) over a very small increment of time (dt). On a graph, this is interpreted as the slope of a curve at a specific point (slope being defined as *rise over run*).

It is also possible to build a controller with proportional and derivative actions, but lacking integral action. These are most commonly used in applications prone to wind-up[7], and where the elimination of offset is not critical:

$$m = K_p e + \tau_d \frac{de}{dt} + b$$

Many PID controllers offer the option of calculating derivative response based on rates of change for the process variable (PV) only, rather than the error (PV − SP or SP − PV). This avoids huge "spikes" in the output of the controller if ever a human operator makes a sudden change in setpoint[8]. The mathematical expression for such a controller would look like this[9]:

$$m = K_p e + \frac{1}{\tau_i} \int e \, dt + \tau_d \frac{d\text{PV}}{dt} + b$$

Even when derivative control action is calculated on PV alone (rather than on error), it is still useful for controlling processes dominated by large lag times. The presence of derivative control action in a PID controller generally means the proportional (P) and integral (I) terms may be adjusted more aggressively than before, since derivative (D) will act to limit overshoot. In other words, the judicious presence of derivative action in a PID controller lets us "get away" with using a bit more P and I action than we ordinarily could, resulting in faster approach to setpoint with minimal overshoot.

It should be mentioned that derivative mode should be used with caution. Since it acts on rates of change, derivative action will "go crazy" if it sees substantial noise in the PV signal. Even small amounts of noise possess extremely large rates of change (defined as percent PV change per minute of time) owing to the relatively high frequency of noise compared to the timescale of physical process changes.

Ziegler and Nichols, the engineers who wrote the ground-breaking paper entitled "Optimum Settings for Automatic Controllers" had these words to say regarding "pre-act" control (page 762 of the November 1942 *Transactions of the A.S.M.E.*):

> The latest control effect made its appearance under the trade name "Pre-Act." On some control applications, the addition of pre-act response made such a remarkable improvement that it appeared to be in embodiment of mythical "anticipatory" controllers. On other applications it appeared to be worse than useless. Only the difficulty of predicting the usefulness and adjustment of this response has kept it from being more widely used.

[7] An example of such an application is where the output of a loop controller may be "de-selected" or otherwise "over-ridden" by some other control function. This sort of control strategy is often used in energy-conserving controls, where multiple controllers monitoring different process variables selectively command a single FCE.

[8] It should not be assumed that such spikes are always undesirable. In processes characterized by long lag times, such a response may be quite helpful in overcoming that lag for the purpose of rapidly achieving new setpoint values. Slave (secondary) controllers in cascaded systems – where the controller receives its setpoint signal from the output of another (primary, or master) controller – may similarly benefit from derivative action calculated on error instead of just PV. As usual, the specific needs of the application dictate the ideal controller configuration.

[9] The expression shown is valid for a direct-acting controller. A reverse-acting controller with derivative action on PV rather than error must *subtract* the derivative term rather than add it to the output value.

2.7 Summary of PID control terms

PID control can be a confusing concept to understand. Here, a brief summary of each term within PID (P. I, and D) is presented for your learning benefit.

2.7.1 Proportional control mode (P)

Proportional – sometimes called *gain* or *sensitivity* – is a control action reproducing changes in input as changes in output. Proportional controller action responds to present changes in input by generating immediate and commensurate changes in output. When you think of "proportional action" (P), think *prompt*: this control action works immediately (never too soon or too late) to match changes in the input signal.

Mathematically defined, proportional action is the ratio of output change to input change. This may be expressed as a quotient of differences, or as a derivative (a rate of change, using calculus notation):

$$\text{Gain value} = \frac{\Delta\text{Output}}{\Delta\text{Input}}$$

$$\text{Gain value} = \frac{d\text{Output}}{d\text{Input}} = \frac{dm}{de}$$

For example, if the PV input of a proportional-only process controller with a gain of 2 suddenly changes ("steps") by 5 percent, and the output will immediately jump by 10 percent ($\Delta\text{Output} = \text{Gain} \times \Delta\text{Input}$). The direction of this output jump in relation to the direction of the input jump depends on whether the controller is configured for direct or reverse action.

A legacy term used to express this same concept is *proportional band*: the mathematical reciprocal of gain. "Proportional band" is defined as the amount of input change necessary to evoke full-scale (100%) output change in a proportional controller. Incidentally, it is always expressed as a percentage, never as fraction or as a per unit value:

$$\text{Proportional Band value} = \frac{\Delta\text{Input}}{\Delta\text{Output}}$$

$$\text{Proportional Band value} = \frac{d\text{Input}}{d\text{Output}} = \frac{de}{dm}$$

Using the same example of a proportional controller exhibiting an output "step" of 10% in response to a PV "step" of 5%, the proportional band would be 50%: the reciprocal of its gain ($\frac{1}{2} = 50\%$). Another way of saying this is that a 50% input "step" would be required to change the output of this controller by a full 100%, since its gain is set to a value of 2.

2.7.2 Integral control mode (I)

Integral – sometimes called *reset* or *floating control* – is a control action causing the output signal to change over time at a rate proportional to the amount of error (the difference between PV and SP values). Integral controller action responds to error accumulated over time, ramping the output signal are far as it needs to go to completely eliminate error. If proportional (P) action tells the output how *far* to move when an error appears, integral (I) action tells the output how *fast* to move when an error appears. If proportional (P) action acts on the *present*, integral (I) action acts on the *past*. Thus, how far the output signal gets driven by integral action depends on the *history* of the error over time: how much error existed, and for how long. When you think of "integral action" (I), think *impatience*: this control action drives the output further and further the longer PV fails to match SP.

Mathematically defined, integral action is the ratio of output *velocity* to input error:

$$\text{Integral value (repeats per minute)} = \frac{\text{Output velocity}}{\text{Input error}}$$

$$\text{Integral value (repeats per minute)} = \frac{\frac{dm}{dt}}{e}$$

An alternate way to express integral action is to use the reciprocal unit of "minutes per repeat." If we define integral action in these terms, the defining equations must be reciprocated:

$$\text{Integral time constant (minutes per repeat)} = \tau_i = \frac{\text{Input error}}{\text{Output velocity}}$$

$$\text{Integral time constant (minutes per repeat)} = \tau_i = \frac{e}{\frac{dm}{dt}}$$

For example, if an error of 5% appears between PV and SP on an integral-only process controller with an integral value of 3 repeats per minute (i.e. an integral time constant of 0.333 minutes per repeat), the output will begin ramping at a rate of 15% per minute ($\frac{dm}{dt}$ = Integral_value $\times$ e, or $\frac{dm}{dt} = \frac{e}{\tau_i}$). In most PI and PID controllers, integral response is also multiplied by proportional gain, so the same conditions applied to a PI controller that happened to also have a gain of 2 would result in an output ramping rate of 30% per minute ($\frac{dm}{dt}$ = Gain_value $\times$ Integral_value $\times$ e, or $\frac{dm}{dt}$ = Gain_value $\times$ $\frac{e}{\tau_i}$). The direction of this ramping in relation to the direction (sign) of the error depends on whether the controller is configured for direct or reverse action.

2.7.3 Derivative control mode (D)

Derivative – sometimes called *rate* or *pre-act* – is a control action causing the output signal to be offset by an amount proportional to the rate at which the input is changing. Derivative controller action responds to how quickly the input changes over time, biasing the output signal commensurate with that rate of input change. If proportional (P) action tells the output how *far* to move when an error appears, derivative (D) action tells the output how far to move when the input *ramps*. If proportional (P) action acts on the *present* and integral (I) action acts on the *past*, derivative (D) action acts on the *future*: it effectively "anticipates" overshoot by tempering the output response according to how fast the process variable is rising or falling. When you think of "derivative action" (D), think *discretion*: this control action is cautious and prudent, working against change.

Mathematically defined, derivative action is the ratio of output offset to input *velocity*:

$$\text{Derivative time constant (minutes)} = \tau_d = \frac{\text{Output offset}}{\text{Input velocity}}$$

$$\text{Derivative time constant (minutes)} = \tau_d = \frac{\Delta\text{Output}}{\frac{de}{dt}}$$

For example, if the PV signal begins to ramp at a rate of 5% per minute on a process controller with a derivative time constant of 4 minutes, the output will immediately become offset by 20% (ΔOutput = Derivative_value $\times$ $\frac{de}{dt}$). In most PD and PID controllers, derivative response is also multiplied by proportional gain, so the same conditions applied to a PD controller that happened to also have a gain of 2 would result in an immediate offset of 40% (ΔOutput = Gain_value $\times$ Derivative_value $\times$ $\frac{de}{dt}$). The direction (sign) of this offset in relation to the direction of the input ramping depends on whether the controller is configured for direct or reverse action.

2.8 Different PID equations

For better or worse, there are no fewer than *three* different forms of PID equations implemented in modern PID controllers: the *parallel*, *ideal*, and *series*. Some controllers offer the choice of more than one equation, while others implement just one. It should be noted that more variations of PID equation exist than these three, but that these are the three major variations.

2.8.1 Parallel PID equation

The equation used to describe PID control so far in this chapter is the simplest form, sometimes called the *parallel* equation, because each action (P, I, and D) occurs in separate terms of the equation, with the combined effect being a simple sum:

$$m = K_p e + \frac{1}{\tau_i} \int e\, dt + \tau_d \frac{de}{dt} + b \qquad \textbf{Parallel PID equation}$$

In the parallel equation, each action parameter (K_p, τ_i, τ_d) is independent of the others. At first, this may seem to be an advantage, for it means each adjustment made to the controller should only affect one aspect of its action. However, there are times when it is better to have the gain parameter affect all three control actions (P, I, and D)[10].

We may show the independence of the three actions mathematically, by breaking the equation up into three different parts, each one describing its contribution to the output (Δm):

$$\Delta m = K_p \Delta e \qquad \text{Proportional action}$$

$$\Delta m = \frac{1}{\tau_i} \int e\, dt \qquad \text{Integral action}$$

$$\Delta m = \tau_d \frac{de}{dt} \qquad \text{Derivative action}$$

As you can see, the three portions of this PID equation are completely separate, with each tuning parameter (K_p, τ_i, and τ_d) acting independently within its own term of the equation.

[10] An example of a case where it is better for gain (K_p) to influence all three control modes is when a technician re-ranges a transmitter to have a larger or smaller span than before, and must re-tune the controller to maintain the same loop gain as before. If the controller's PID equation takes the parallel form, the technician must adjust the P, I, *and* D tuning parameters proportionately. If the controller's PID equation uses K_p as a factor in all three modes, the technician need only adjust K_p to re-stabilize the loop.

2.8.2 Ideal PID equation

An alternate version of the PID equation designed such that the gain (K_p) affects all three actions is called the *Ideal* or *ISA* equation:

$$m = K_p \left(e + \frac{1}{\tau_i} \int e \, dt + \tau_d \frac{de}{dt} \right) + b \qquad \textbf{Ideal or ISA PID equation}$$

Here, the gain constant (K_p) is distributed to all terms within the parentheses, equally affecting all three control actions. Increasing K_p in this style of PID controller makes the P, the I, *and* the D actions equally more aggressive.

We may show this mathematically, by breaking the "ideal" equation up into three different parts, each one describing its contribution to the output (Δm):

$$\Delta m = K_p \Delta e \qquad \text{Proportional action}$$

$$\Delta m = \frac{K_p}{\tau_i} \int e \, dt \qquad \text{Integral action}$$

$$\Delta m = K_p \tau_d \frac{de}{dt} \qquad \text{Derivative action}$$

As you can see, all three portions of this PID equation are influenced by the gain (K_p) owing to algebraic distribution, but the integral and derivative tuning parameters (τ_i and τ_d) act independently within their own terms of the equation.

2.8.3 Series PID equation

A third version, with origins in the peculiarities of pneumatic controller mechanisms and analog electronic circuits, is called the *Series* or *Interacting* equation:

$$m = K_p \left[\left(\frac{\tau_d}{\tau_i} + 1 \right) e + \frac{1}{\tau_i} \int e \, dt + \tau_d \frac{de}{dt} \right] + b \qquad \textbf{Series or Interacting PID equation}$$

Here, the gain constant (K_p) affects all three actions (P, I, and D) just as with the "ideal" equation. The difference, though, is the fact that both the integral and derivative constants have an effect on proportional action as well! That is to say, adjusting either τ_i or τ_d does not merely adjust those actions, but also influences the aggressiveness of proportional action[11].

We may show this mathematically, by breaking the "series" equation up into three different parts, each one describing its contribution to the output (Δm):

$$\Delta m = K_p \left(\frac{\tau_d}{\tau_i} + 1 \right) \Delta e \qquad \text{Proportional action}$$

$$\Delta m = \frac{K_p}{\tau_i} \int e \, dt \qquad \text{Integral action}$$

$$\Delta m = K_p \tau_d \frac{de}{dt} \qquad \text{Derivative action}$$

As you can see, all three portions of this PID equation are influenced by the gain (K_p) owing to algebraic distribution. However, the proportional term is also affected by the values of the integral and derivative tuning parameters (τ_i and τ_d). Therefore, adjusting τ_i affects both the I and P actions, adjusting τ_d affects both the D and P actions, and adjusting K_p affects all three actions.

This "interacting" equation is an artifact of certain pneumatic and electronic controller designs. Back when these were the dominant technologies, and PID controllers were modularly designed such that integral and derivative actions were separate hardware modules included in a controller at additional cost beyond proportional-only action, the easiest way to implement the integral and derivative actions was in a way that just happened to have an interactive effect on controller gain. In other words, this odd equation form was a sort of compromise made for the purpose of simplifying the physical design of the controller.

Interestingly enough, many digital PID controllers are programmed to implement the "interacting" PID equation even though it is no longer an artifact of controller hardware. The rationale for this programming is to have the digital controller behave identically to the legacy analog electronic or pneumatic controller it is replacing. This way, the proven tuning parameters of the old controller may be plugged into the new digital controller, yielding the same results. In essence, this is a form of "backward compatibility" between digital PID control and analog (electronic or pneumatic) PID control.

[11] This becomes especially apparent when using derivative action with low values of τ_i (aggressive integral action). The error-multiplying term $\frac{\tau_d}{\tau_i} + 1$ may become quite large if τ_i is small, even with modest τ_d values.

2.9 Analog electronic PID controllers

Although analog electronic process controllers are considered a newer technology than pneumatic process controllers, they are actually "more obsolete" than pneumatic controllers. Panel-mounted (inside a control room environment) analog electronic controllers were a great improvement over panel-mounted pneumatic controllers when they were first introduced to industry, but they were superseded by digital controller technology later on. Field-mounted pneumatic controllers were either replaced by panel-mounted electronic controllers (either analog or digital) or left alone. Applications still exist for field-mounted pneumatic controllers, even now at the beginning of the 21^{st} century, but very few applications exist for analog electronic controllers in any location.

Analog electronic controllers enjoy two inherent advantages over digital electronic controllers: greater reliability[12] and faster response. However, these advantages have been diminishing as digital control technology has advanced. Today's digital electronic technology is far more reliable than the digital technology available during the heyday of analog electronic controllers. Now that digital controls have achieved very high levels of reliability, the first advantage of analog control is largely academic[13], leaving only the second advantage for practical consideration. The advantage of faster speed may be fruitful in applications such as motion control, but for most industrial processes even the slowest digital controller is fast enough[14]. Furthermore, the numerous advantages offered by digital technology (data recording, networking capability, self-diagnostics, flexible configuration, function blocks for implementing different control strategies) severely weaken the relative importance of reliability and speed.

Most analog electronic PID controllers utilize *operational amplifiers* in their designs. It is relatively easy to construct circuits performing amplification (gain), integration, differentiation, summation, and other useful control functions with just a few opamps, resistors, and capacitors.

[12] The reason for this is the low component count compared to a comparable digital control circuit. For any given technology, a simpler device will tend to be more reliable than a complex device if only due to there being fewer components to fail. This also suggests a third advantage of analog controllers over digital controllers, and that is the possibility of easily designing and constructing your own for some custom application such as a hobby project. A digital controller is not outside the reach of a serious hobbyist to design and build, but it is definitely more challenging due to the requirement of programming expertise in addition to electronic hardware expertise.

[13] It is noteworthy that analog control systems are completely immune from "cyber-attacks" (malicious attempts to foil the integrity of a control system by remote access), due to the simple fact that their algorithms are fixed by physical laws and properties of electronic components rather than by code which may be edited. This new threat constitutes an inherent weakness of digital technology, and has spurred some thinkers in the field to reconsider analog controls for the most critical applications.

[14] The real problem with digital controller speed is that the time delay between successive "scans" translates into dead time for the control loop. Dead time is the single greatest impediment to feedback control.

2.9.1 Proportional control action

The basic proportional-only control algorithm follows this formula:

$$m = K_p e + b$$

Where,
m = Controller output
e = Error (difference between PV and SP)
K_p = Proportional gain
b = Bias

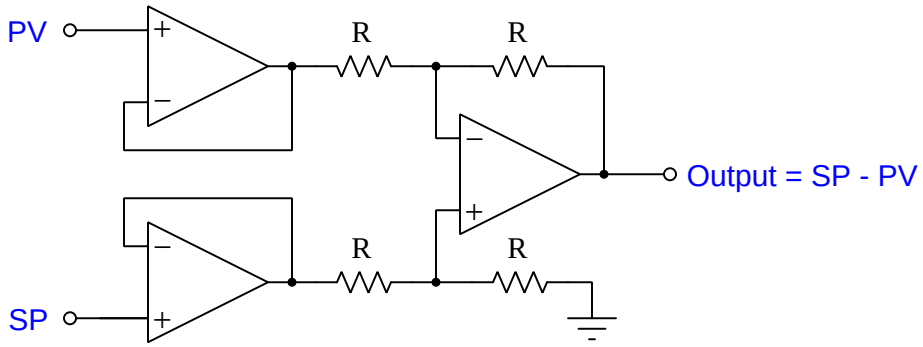
The "error" variable (e) is the mathematical difference between process variable and setpoint. If the controller is direct-acting, $e = \text{PV} - \text{SP}$. If the controller is reverse-acting, $e = \text{SP} - \text{PV}$. Thus,

$$m = K_p(\text{PV} - \text{SP}) + b \qquad \text{Direct-acting}$$

$$m = K_p(\text{SP} - \text{PV}) + b \qquad \text{Reverse-acting}$$

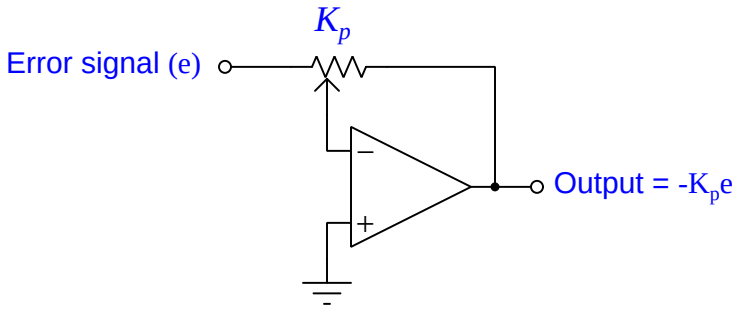
Mathematical operations such as subtraction, multiplication by a constant, and addition are quite easy to perform using analog electronic (operational amplifier) circuitry. Prior to the advent of reliable digital electronics for industrial applications, it was natural to use analog electronic circuitry to perform proportional control for process control loops.

For example, the subtraction function necessary to calculate error (e) from process variable and setpoint signals may be performed with a three-amplifier "subtractor" circuit:



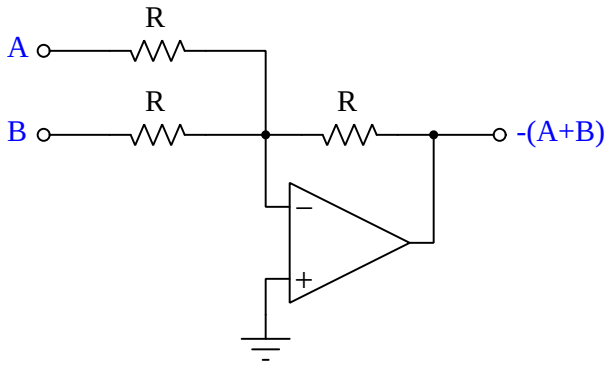
This particular subtractor circuit calculates error for a reverse-acting controller. As the PV signal increases, the error signal decreases (becomes more negative). It could be modified for direct action simply by swapping the two inputs: SP on top and PV on bottom such that the Output becomes PV − SP.

Gain is really nothing more than multiplication by a constant, in this case the constant being K_p. A very simple one-amplifier analog circuit for performing this multiplication is the *inverting*[15] *amplifier* circuit:



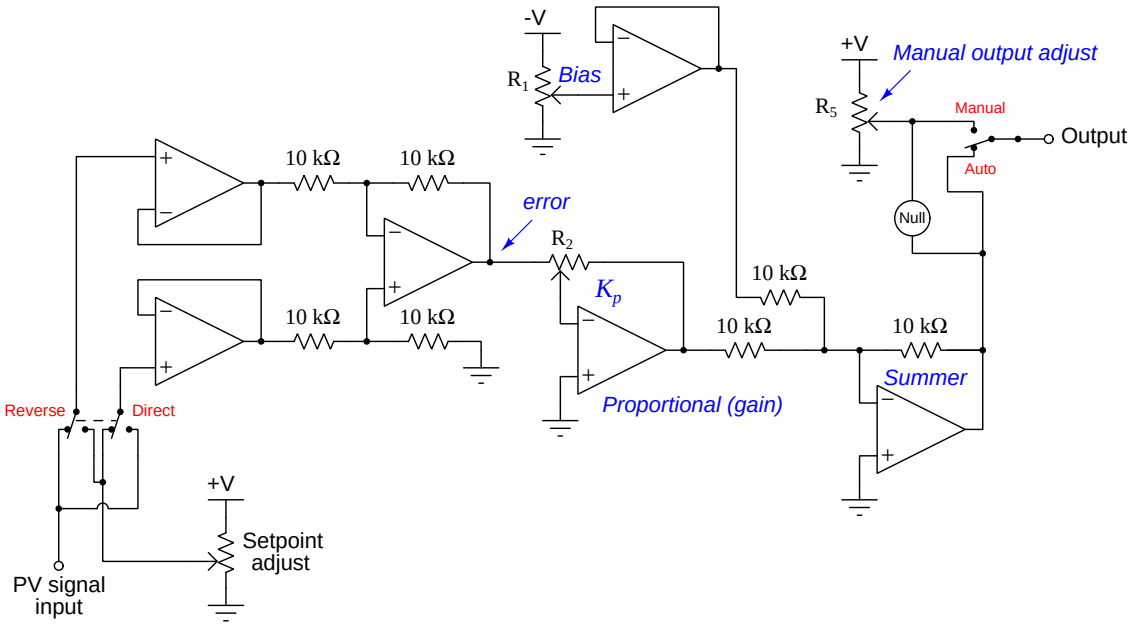
With the potentiometer's wiper in mid-position, the voltage gain of this circuit will be 1 (with an inverted polarity which we shall ignore for now). Moving the wiper toward the left-hand side of the potentiometer increases the circuit's gain past unity, while moving the wiper toward the right-hand side of the potentiometer decreases the gain toward zero.

In order to add the bias (b) term in the proportional control equation, we need an analog circuit capable of summing two voltage signals. This need is nicely met in the *inverting summer* circuit, shown here:



[15] This circuit configuration is called "inverting" because the mathematical sign of the output is always opposite that of the input. This sign inversion is not an intentional circuit feature, but rather a consequence of the input signal facing the opamp's inverting input. Non-inverting multiplier circuits also exist, but are more complicated when built to achieve multiplication factors less than one.

Combining all these analog functions together into one circuit, and adding a few extra features such as direct/reverse action selection, bias adjustment, and manual control with a null voltmeter to facilitate bumpless mode transfer, gives us this complete analog electronic proportional controller:



2.9.2 Derivative and integral control actions

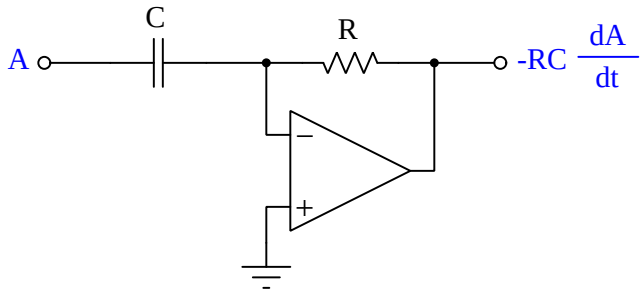
Differentiating and integrating live voltage signals with respect to time is quite simple using operational amplifier circuits. Instead of using all resistors in the negative feedback network, we may implement these calculus functions by using a combination of *capacitors* and resistors, exploiting the capacitor's natural derivative relationship between voltage and current:

$$I = C\frac{dV}{dt}$$

Where,

I = Current through the capacitor (amperes)
C = Capacitance of capacitor (farads)
V = Voltage across the capacitor (volts)
$\frac{dV}{dt}$ = Rate-of-change of voltage across the capacitor (volts per second)

If we build an operational amplifier with a resistor providing negative feedback current through a capacitor, we create a *differentiator* circuit where the output voltage is proportional to the rate-of-change of the input voltage:



Since the inverting input of the operational amplifier is held to ground potential by feedback (a "virtual ground"), the capacitor experiences the full input voltage of signal A. So, as A varies over time, the current through that capacitor will directly represent the signal A's rate of change over time ($I = C\frac{dA}{dt}$). This current passes through the feedback resistor, creating a voltage drop at the output of the amplifier directly proportional to signal A's rate of change over time. Thus, the output voltage of this circuit reflects the input voltage's instantaneous rate of change, albeit with an inverted polarity. The mathematical term RC is the *time constant* of this circuit. For a differentiator circuit such as this, we typically symbolize its time constant as τ_d (the "derivative" time constant).

For example, if the input voltage to this differentiator circuit were to ramp at a constant rate of +4.3 volts per second (rising) with a resistor value of 10 kΩ and a capacitor value of 33 μF (i.e. τ_d = 0.33 seconds), the output voltage would be a constant −1.419 volts:

$$V_{out} = -RC\frac{dV_{in}}{dt}$$

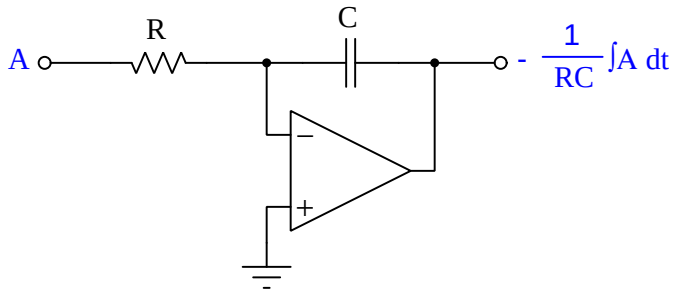
$$V_{out} = -(10000\ \Omega)(33 \times 10^{-6}\ \text{F})\left(\frac{4.3\ \text{V}}{\text{s}}\right)$$

$$V_{out} = -(0.33\ \text{s})\left(\frac{4.3\ \text{V}}{\text{s}}\right)$$

$$V_{out} = -1.419\ \text{V}$$

Recall that the purpose of derivative action in a PID controller is to react to sudden changes in either the error (e) or the process variable (PV). This circuit fulfills that function, by generating an output proportional to the input voltage's rate of change.

If we simply swap[16] the locations of the resistor and capacitor in the feedback network of this operational amplifier circuit, we create an *integrator* circuit where the output voltage rate-of-change is proportional to the input voltage:



This integrator circuit provides the exact inverse function of the differentiator. Rather than a changing input signal generating an output signal proportional to the input's rate of change, an input signal in this circuit controls the rate at which the output signal changes.

The way it works is by acting as a *current source*, pumping current into the capacitor at a value determined by the input voltage and the resistor value. Just as in the previous (differentiator) circuit where the inverting terminal of the amplifier was a "virtual ground" point, the input voltage in this circuit is impressed across the resistor R. This creates a current which must go through capacitor C on its way either to or from the amplifier's output terminal. As we have seen in the capacitor's equation ($I = C\frac{dV}{dt}$), a current forced through a capacitor causes the capacitor's voltage to change over time. This changing voltage becomes the output signal of the integrator circuit. As in the case of the differentiator circuit, the mathematical term RC is the *time constant* of this circuit as well. Being an integrator, we customarily represent this "integral" time constant as τ_i.

Any amount of change in output voltage (ΔV_{out}) occurring between some initial time (t_0) and a finishing time (t_f) may be calculated by the following integral:

$$\Delta V_{out} = -\frac{1}{RC}\int_{t_0}^{t_f} V_{in}\, dt$$

If we wish to know the absolute output voltage at the end of that time interval, all we need to do is add the circuit's initial output voltage (V_0, i.e. the voltage stored in the capacitor at the initial time t_0) to the calculated change:

$$V_{out} = -\frac{1}{RC}\int_{t_0}^{t_f} V_{in}\, dt + V_0$$

[16] This inversion of function caused by the swapping of input and feedback components in an operational amplifier circuit points to a fundamental principle of negative feedback networks: namely, that placing a mathematical element within the feedback loop causes the amplifier to exhibit the inverse of that element's intrinsic function. This is why voltage dividers placed within the feedback loop cause an opamp to have a multiplicative gain (division → multiplication). A circuit element exhibiting a logarithmic response, when placed within a negative feedback loop, will cause the amplifier to exhibit an exponential response (logarithm → exponent). Here, an element having a time-differentiating response, when placed inside the feedback loop, causes the amplifier to time-integrate (differentiation → integration). Since the opamp's output voltage must assume any value possible to maintain (nearly) zero differential voltage at the input terminals, placing a mathematical function in the feedback loop forces the output to assume the inverse of that function in order to "cancel out" its effects and achieve balance at the input terminals.

For example, if we were to input a constant DC voltage of +1.7 volts to this circuit with a resistor value of 81 kΩ and a capacitor value of 47 μF (i.e. $\tau_i = 3.807$ seconds), the output voltage would ramp at a constant rate of −0.447 volts per second[17]. If the output voltage were to begin at −3.0 volts and be allowed to ramp for exactly 12 seconds at this rate, it would reach a value of −8.359 volts at the conclusion of that time interval:

$$V_{out} = -\frac{1}{RC}\int_{t_0}^{t_f} V_{in}\, dt + V_0$$

$$V_{out} = -\left(\frac{1}{(81000\,\Omega)(47 \times 10^{-6}\text{ F})}\right)\left(\int_0^{12} 1.7\text{ V}\, dt\right) - 3\text{ V}$$

$$V_{out} = -\left(\frac{1}{3.807\text{ s}}\right)(20.4\text{ V} \cdot \text{s}) - 3\text{ V}$$

$$V_{out} = -5.359\text{ V} - 3\text{ V}$$

$$V_{out} = -8.359\text{ V}$$

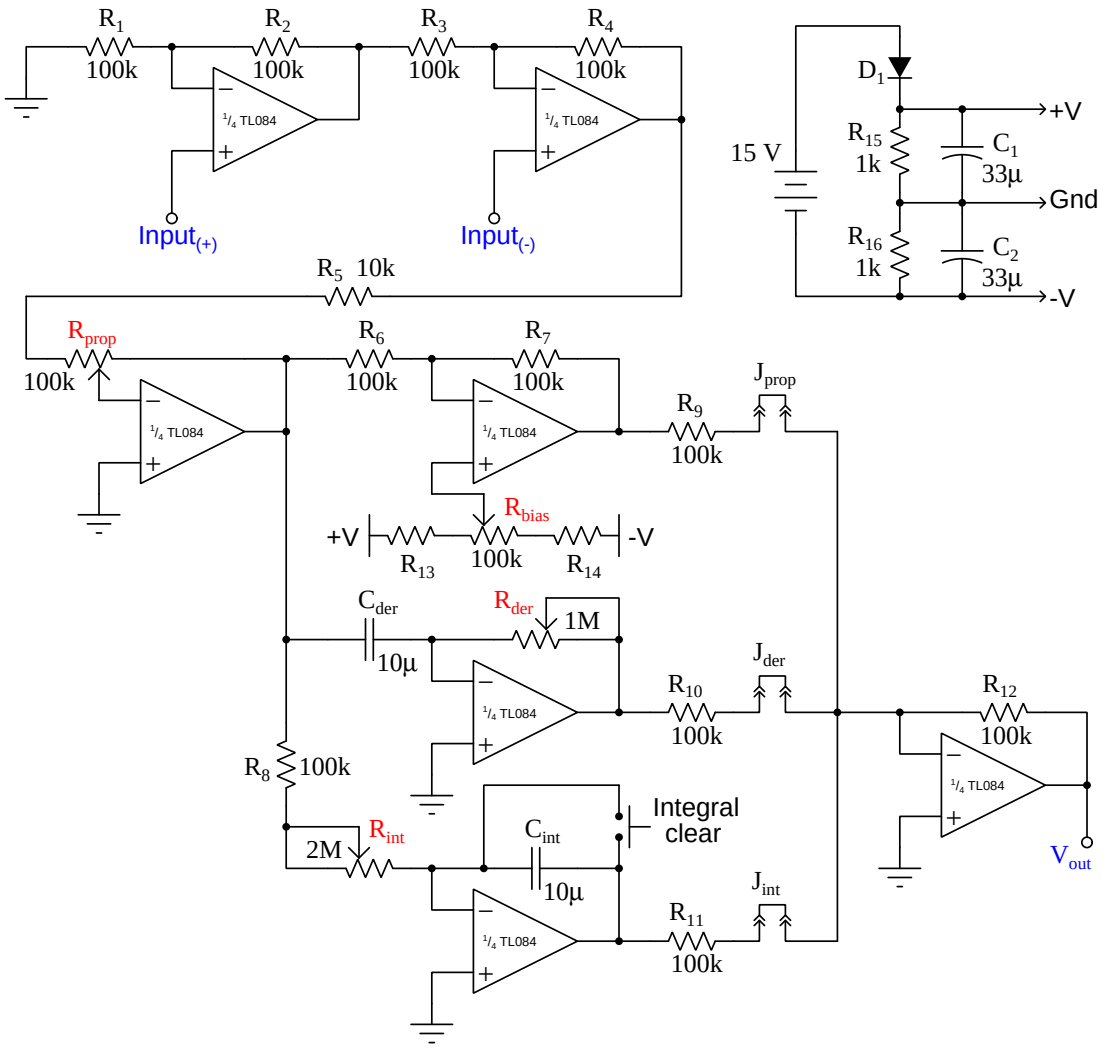
If, after ramping for some amount of time, the input voltage of this integrator circuit is brought to zero, the integrating action will cease. The circuit's output will simply hold at its last value until another non-zero input signal voltage appears.

Recall that the purpose of integral action in a PID controller is to eliminate offset between process variable and setpoint by calculating the error-time product (how far PV deviates from SP, and for how long). This circuit will fulfills that function if the input voltage is the error signal, and the output voltage contributes to the output signal of the controller.

[17] If this is not apparent, imagine a scenario where the +1.7 volt input existed for precisely one second's worth of time. However much the output voltage ramps in that amount of time must therefore be its rate of change in volts per second (assuming a linear ramp). Since we know the area accumulated under a constant value of 1.7 (high) over a time of 1 second (wide) must be 1.7 volt-seconds, and τ_i is equal to 3.807 seconds, the integrator circuit's output voltage must ramp 0.447 volts during that interval of time. If the input voltage is positive and we know this is an inverting opamp circuit, the direction of the output voltage's ramping must be negative, thus a ramping rate of *−0.447 volts per second*.

2.9.3 Full-PID circuit design

The following schematic diagram shows a full PID controller implemented using eight operational amplifiers, designed to input and output voltage signals representing PV, SP, and Output[18]:



It is somewhat stunning to realize that such a controller, fully capable of controlling many industrial process types, may be constructed using only two integrated circuit "chips" (two "quad" operational amplifiers) and a handful of passive electronic components. The only significant engineering challenge in this simple circuit design is achieving slow enough time constants (in the

[18]The two input terminals shown, $\text{Input}_{(+)}$ and $\text{Input}_{(-)}$ are used as PV and SP signal inputs, the correlation of each depending on whether one desires direct or reverse controller action.

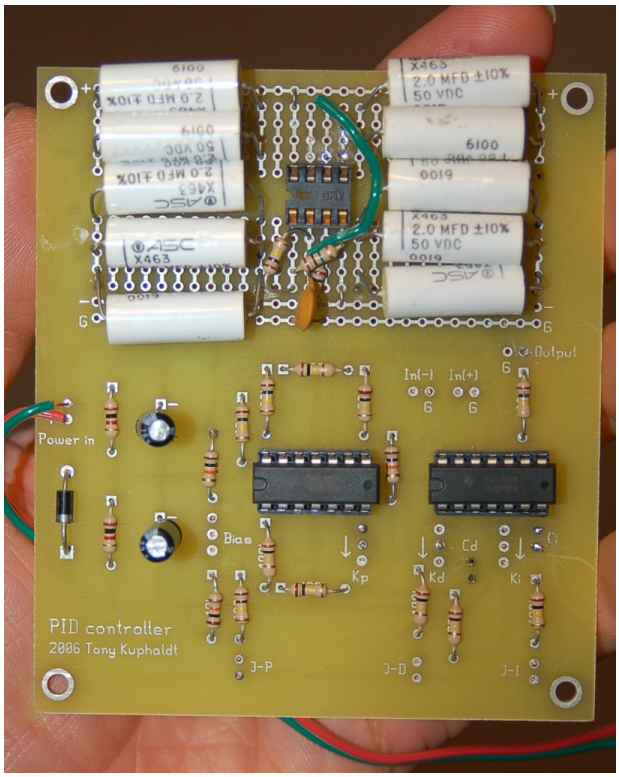
range of minutes rather than seconds) in the integrator and differentiator functions using non-polarized capacitors[19].

This controller implements the so-called *ideal* PID algorithm, with the proportional (gain) value distributing to the integral and derivative terms:

$$m = K_p \left(e + \frac{1}{\tau_i} \int e \, dt + \tau_d \frac{de}{dt} \right) \qquad \textbf{Ideal PID equation}$$

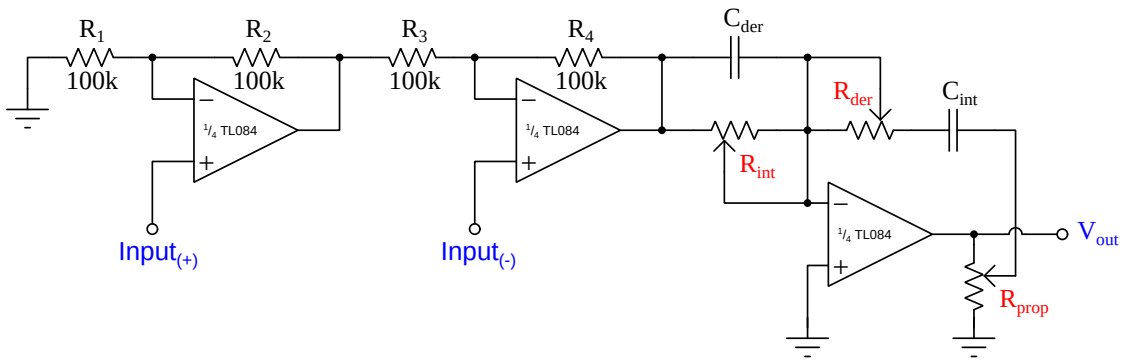
We may determine this from the schematic diagram by noting that the I and D functions each receive their input signals from the output of the proportional amplifier (the one with the R_{prop} potentiometer). Adjusting R_{prop} affects not only the controller's proportional gain, but also the sensitivity of τ_i and τ_d.

An actual implementation of this PID controller in printed circuit board form appears here:



[19] This particular design has integral and derivative time value limits of 10 seconds, maximum. These relatively "quick" tuning values are the result of having to use non-polarized capacitors in the integrator and differentiator stages. The practical limits of cost and size restrict the maximum value of on-board capacitance to around 10 μF each.

It is possible to construct an analog PID controller with fewer components. An example is shown here:



As you can see, a *single* operational amplifier does all the work of calculating proportional, integral, *and* derivative responses. The first two amplifiers do nothing but buffer the input signals and calculate error (PV − SP, or SP − PV, depending on the direction of action).

One of the consequences of consolidating all three control terms in a single amplifier is that those control terms interact with each other. The mathematical expression of this control action is shown here, called the *series* or *interacting* PID equation:

$$m = K_p \left[\left(\frac{\tau_d}{\tau_i} + 1 \right) e + \frac{1}{\tau_i} \int e\, dt + \tau_d \frac{de}{dt} \right] \qquad \textbf{Series or Interacting PID equation}$$

Not only does a change in gain (K_p) alter the relative responses of integral and derivative in the series equation (as it also does in the ideal equation), but changes in either integral or derivative time constants also have an effect on proportional response! This is especially noticeable when the integral time constant is set to some very small value, which is typically the case on fast-responding, self-regulating processes such as liquid flow or liquid pressure control.

It should be apparent that an analog controller implementing the series equation is simpler in construction than one implementing either the parallel or ideal PID equation. This also happens to be true for pneumatic PID controller mechanisms: the simplest analog controller designs all implement the series PID equation[20].

[20] An interesting example of engineering tradition is found in electronic PID controller designs. While it is not too terribly difficult to build an analog electronic controller implementing either the parallel or ideal PID equation (just a few more parts are needed), it is quite challenging to do the same in a pneumatic mechanism. When analog electronic controllers were first introduced to industry, they were often destined to replace old pneumatic controllers. In order to ease the transition from pneumatic to electronic control, manufacturers built their new electronic controllers to behave exactly the same as the old pneumatic controllers they would be replacing. The same legacy followed the advent of digital electronic controllers: many digital controllers were programmed to behave in the same manner as the old pneumatic controllers, for the sake of operational familiarity, not because it was easier to design a digital controller that way.

2.10 Digital PID algorithms

Instrument technicians should not have to concern themselves over the programming details internal to digital PID controllers. Ideally, a digital PID controller should simply perform the task of executing PID control with all the necessary features (setpoint tracking, output limiting, etc.) without the end-user having to know anything about those details. However, in my years of experience I have seen enough examples of poor PID implementation to warrant an explanatory section in this book, both so instrumentation professionals may recognize poor PID implementation when they see it, and also so those with the responsibility of designing PID algorithms may avoid some common mistakes.

2.10.1 Position versus velocity algorithms

The canonical "ideal" or "ISA" variety of PID equation takes the following form:

$$m = K_p \left(e + \frac{1}{\tau_i} \int e \, dt + \tau_d \frac{de}{dt} \right)$$

Where,

m = Controller output
e = Error (SP − PV or PV − SP, depending on controller action being direct or reverse)
K_p = Controller gain
τ_i = Integral (reset) time constant
τ_d = Derivative (rate) time constant

The same equation may be written in terms of "gains" rather than "time constants" for the integral and derivative terms. This re-writing exhibits the advantage of consistency from the perspective of PID tuning, where each tuning constant has the same (increasing) effect as its numerical value grows larger:

$$m = K_p \left(e + K_i \int e \, dt + K_d \frac{de}{dt} \right)$$

Where,

m = Controller output
e = Error
K_p = Controller gain
K_i = Integral (reset) gain (repeats per unit time)
K_d = Derivative (rate) gain

However the equation is written, there are two major ways in which it is commonly implemented in a digital computer. One way is the *position* algorithm, where the result of each pass through the program "loop" calculates the actual output value. If the final control element for the loop is a control valve, this value will be the position of that valve's stem, hence the name *position algorithm*. The other way is the so-called *velocity* algorithm, where the result of each pass through the program "loop" calculates the amount the output value will *change*. Assuming a control valve for the final control element once again, the value calculated by this algorithm is the distance the valve stem will travel *per scan of the program*. In other words, the magnitude of this value describes how *fast* the valve stem will travel, hence the name *velocity algorithm*.

Mathematically, the distinction between the position and velocity algorithms is a matter of differentials: the position equation solves for the output value (m) directly while the velocity equation solves for small increments (differentials) of m, or dm.

A comparison of the position and velocity equations shows both the similarities and the differences:

$$m = K_p \left(e + K_i \int e\, dt + K_d \frac{de}{dt} \right) \qquad \text{Position equation}$$

$$dm = K_p \left(de + K_i e\, dt + K_d \frac{d^2 e}{dt} \right) \qquad \text{Velocity equation}$$

Of the two approaches to implementing PID control, the position algorithm makes the most intuitive sense and is the easiest to understand.

We will begin our exploration of both algorithms by examining their application to proportional-only control. This will be a simpler and "gentler" introduction than showing how to implement full PID control. The two respective proportional-only control equations we will consider are shown here:

$$m = K_p e + \text{Bias} \qquad \text{Position equation for P-only control}$$

$$dm = K_p de \qquad \text{Velocity equation for P-only control}$$

You will notice how a "bias" term is required in the position equation to keep track of the output's "starting point" each time a new output value is calculated. No such term is required in the velocity equation, because the computer merely calculates *how far the output moves from its last value* rather than the output's value from some absolute reference.

First, we will examine a simple pseudocode program for the "position" equation form:

Pseudocode listing for a "position algorithm" proportional-only controller

```
DECLARE PV, SP, and Out to be floating-point variables
DECLARE K_p, Error, and Bias to be floating-point variables
DECLARE Action, and Mode to be boolean variables

LOOP
  SET PV = analog_input_channel_N  // Update PV
  SET K_p = operator_input_channel_Gain   // From operator interface

  IF Action = 1 THEN
    SET Error = SP - PV      // Calculate error assuming reverse action
  ELSE THEN
    SET Error = PV - SP      // Calculate error assuming direct action
  ENDIF

  IF Mode = 1 THEN       // Automatic mode (if Mode = 1)
    SET Out = K_p * Error + Bias
    SET SP = operator_input_channel_SP   // From operator interface
  ELSE THEN              // Manual mode (if Mode = 0)
    SET Out = operator_input_channel_Out // From operator interface
    SET SP = PV     // Setpoint tracking
    SET Bias = Out  // Output tracking
  ENDIF
ENDLOOP
```

The first SET instructions within the loop update the PV to whatever value is being measured by the computer's analog input channel (channel N in this case), and the K_p variable to whatever value is entered by the human operator through the use of a keypad, touch-screen interface, or networked computer. Next, a set of IF/THEN conditionals determines which way the error should be calculated: Error = SP − PV if the control action is "reverse" (Action = 1) and Error = PV − SP if the control action is "direct" (Action = 0).

The next set of conditional instructions determines what to do in automatic versus manual modes. In automatic mode (Mode = 1), the output value is calculated according to the position equation and the setpoint comes from a human operator's input. In manual mode (Mode = 0), the output value is no longer calculated by an equation but rather is obtained from the human operator's input, the setpoint is forced equal to the process variable, and the Bias value is continually made equal to the value of the output. Setting SP = PV provides the convenient feature of *setpoint tracking*, ensuring an initial error value of zero when the controller is switched back to automatic mode. Setting the Bias equal to the output provides the essential feature of *output tracking*, where the controller begins automatic operation at an output value precisely equal to the last manual-mode output value.

Next, we will examine a simple pseudocode program for the "velocity" equation form:

Pseudocode listing for a "velocity algorithm" proportional-only controller

```
DECLARE PV, SP, and Out to be floating-point variables
DECLARE K_p, Error, and last_Error to be floating-point variables
DECLARE Action, and Mode to be boolean variables

LOOP
  SET PV = analog_input_channel_N  // Update PV
  SET K_p = operator_input_channel_Gain   // From operator interface
  SET last_Error = Error

  IF Action = 1 THEN
    SET Error = SP - PV      // Calculate error assuming reverse action
  ELSE THEN
    SET Error = PV - SP      // Calculate error assuming direct action
  ENDIF

  IF Mode = 1 THEN       // Automatic mode (if Mode = 1)
    SET Out = Out + (K_p * (Error - last_Error))
    SET SP = operator_input_channel_SP   // From operator interface
  ELSE THEN              // Manual mode (if Mode = 0)
    SET Out = operator_input_channel_Out // From operator interface
    SET SP = PV   // Setpoint tracking
  ENDIF
ENDLOOP
```

The code for the velocity algorithm is mostly identical to the code for the position algorithm, with just a few minor changes. The first difference we encounter in reading the code from top to bottom is that we calculate a new variable called "last_Error" immediately prior to calculating a new value for Error. The reason for doing this is to provide a way to calculate the differential *change* in error (de) from scan to scan of the program. The variable "last_Error" remembers the value of Error during the previous scan of the program. Thus, the expression "Error − last_Error" is equal to the amount the error has changed from last scan to the present scan.

When the time comes to calculate the output value in automatic mode, we see the SET command calculating the change in output (K_p multiplied by the change in error), then adding this change in output to the existing output value to calculate a new output value. This is how the program translates calculated output increments into an actual output value to drive a final control element. The mathematical expression "K_p * (Error − last_Error)" defines the incremental change in output value, and this increment is then added to the current output value to generate a new output value.

From a human operator's point of view, the position algorithm and the velocity algorithm are

identical with one exception: how each controller reacts to a sudden change in gain (K_p). To understand this difference, let us perform a "thought experiment" where we imagine a condition of constant error between PV and SP. Suppose the controller is operating in automatic mode, with a setpoint of 60% and a (steady) process variable value of 57%. We should not be surprised that a constant error might exist for a proportional-only controller, since we should be well aware of the phenomenon of *proportional-only offset*.

How will this controller react if the gain is suddenly increased in value while operating in automatic mode? If the controller executes the position algorithm, the result of a sudden gain change will be a sudden change in its output value, since output is a direct function of error and gain. However, if the controller executes the velocity algorithm, the result of a sudden gain change will be no change to the output at all, so long as the error *remains constant*. Only when the error begins to change will there be any noticeable difference in the controller's behavior compared to how it acted before the gain change. This is because the velocity algorithm is a function of gain and *change in error*, not error directly.

Comparing the two responses, the velocity algorithm's response to changes in gain is regarded as "better-mannered" than the position algorithm's response to changes in gain. When tuning a controller, we would rather not have the controller's output suddenly jump in response to simple gain changes[21], and so the velocity algorithm is generally preferred. If we allow the gain of the algorithm to be set by another process variable[22], the need for "stable" gain-change behavior becomes even more important.

[21] It should be noted that this is precisely what happens when you change the gain in a pneumatic or an analog electronic controller, since all analog PID controllers implement the "position" equation. Although the choice between "position" and "velocity" algorithms in a digital controller is arbitrary, it is *much* easier to build an analog mechanism or circuit implementing the position algorithm than it is to build an analog "velocity" controller.

[22] We call this an *adaptive gain* control system.

Chapter 3

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.

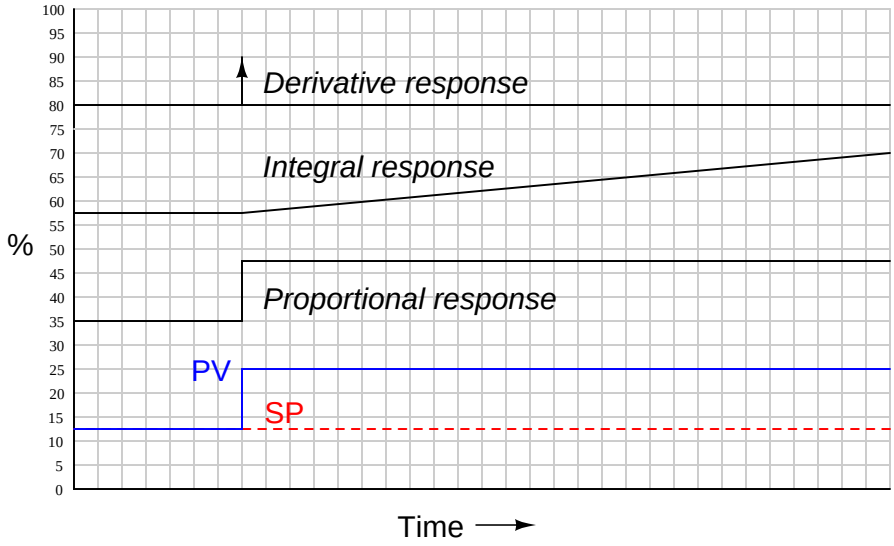
3.1 P, I, and D responses graphed

A very helpful method for understanding the operation of proportional, integral, and derivative control terms is to analyze their respective responses to the same input conditions over time. This section is divided into subsections showing P, I, and D responses for several different input conditions, in the form of graphs. In each graph, the controller is assumed to be *direct-acting* (i.e. an increase in process variable results in an increase in output).

It should be noted that these graphic illustrations are all qualitative, not quantitative. There is too little information given in each case to plot exact responses. The illustrations of P, I, and D actions focus only on the *shapes* of the responses, not their exact numerical values.

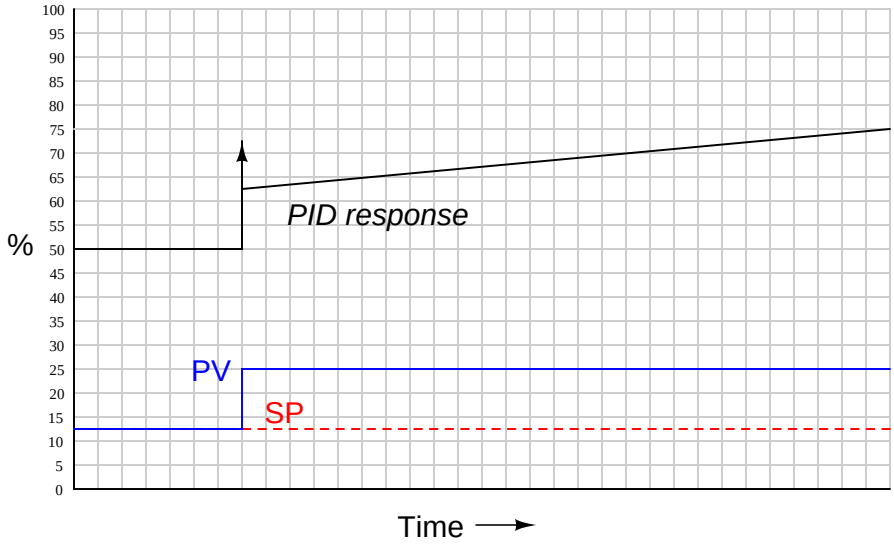
In order to *quantitatively* predict PID controller responses, one would have to know the values of all PID settings, as well as the original starting value of the output before an input change occurred and a time index of when the change(s) occurred.

3.1.1 Responses to a single step-change



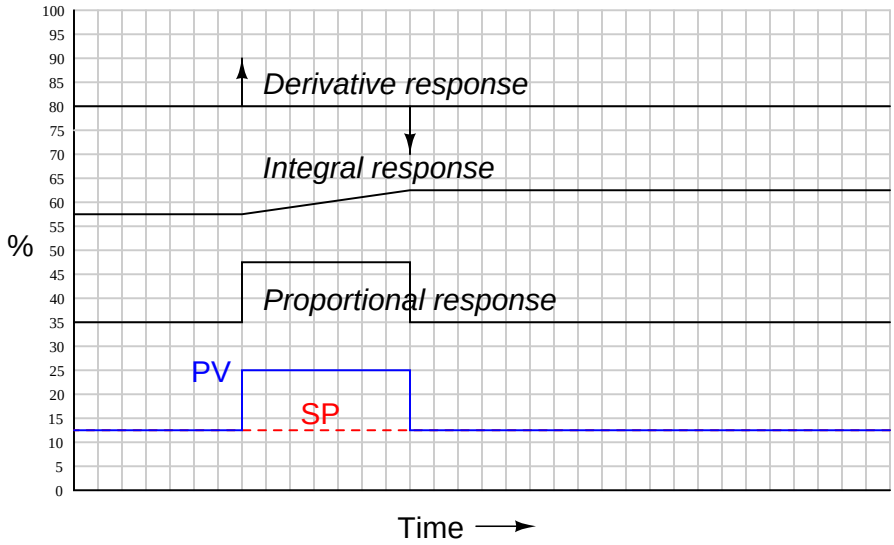
Proportional action directly mimics the shape of the input change (a step). Integral action ramps at a rate proportional to the magnitude of the input step. Since the input step holds a constant value, the integral action ramps at a constant rate (a constant *slope*). Derivative action interprets the step as an *infinite* rate of change, and so generates a "spike[1]" driving the output to saturation.

When combined into one PID output, the three actions produce this response:



[1] This is the meaning of the vertical-pointing arrowheads shown on the trend graph: momentary saturation of the output all the way up to 100%.

3.1.2 Responses to a momentary step-and-return

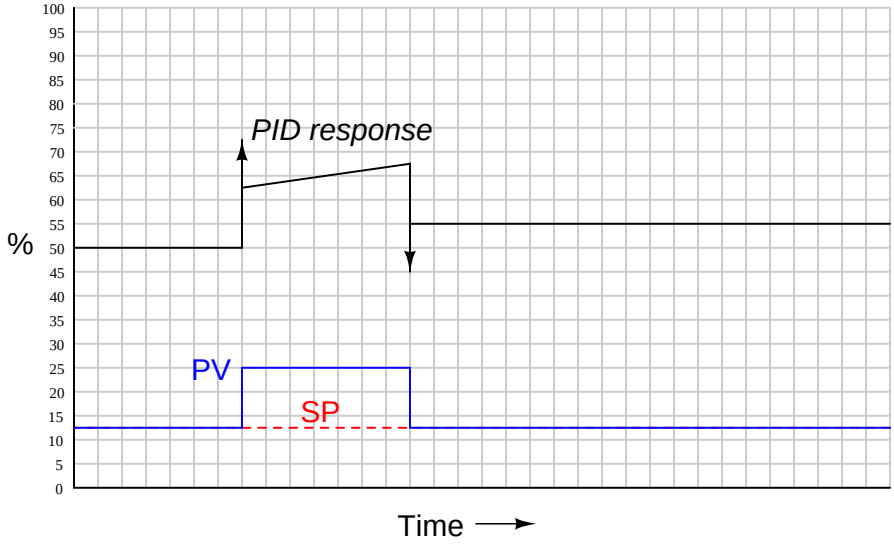


Proportional action directly mimics the shape of the input change (an up-and-down step). Integral action ramps at a rate proportional to the magnitude of the input step, for as long as the PV is unequal to the SP. Once PV = SP again, integral action stops ramping and simply holds the last value[2]. Derivative action interprets both steps as *infinite* rates of change, and so generates "spikes[3]" at the leading and at the trailing edges of the step. Note how the leading (rising) edge causes derivative action to saturate high, while the trailing (falling) edge causes it to saturate low.

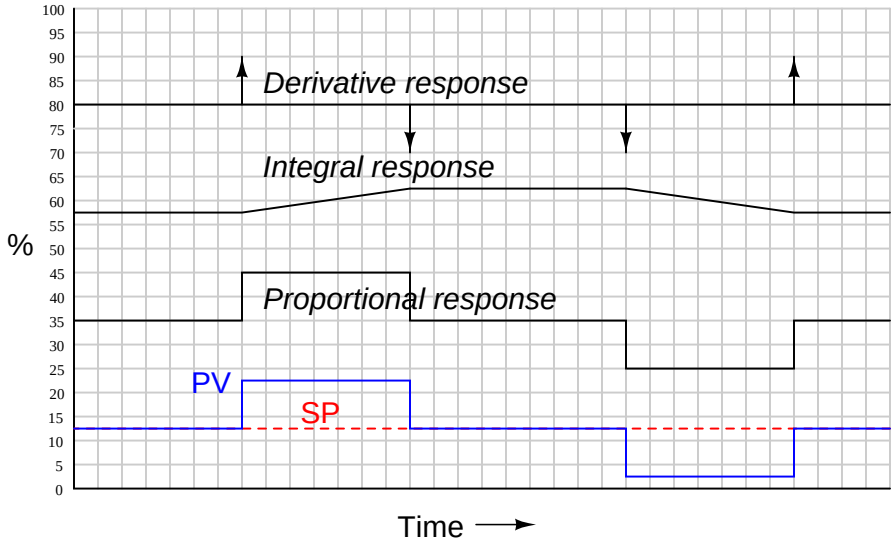
[2]This is a good example of how integral controller action represents the *history* of the PV − SP error. The continued offset of integral action from its starting point "remembers" the area accumulated under the rectangular "step" between PV and SP. This offset will go away only if a *negative* error appears having the same percent-minute product (area) as the positive error step.

[3]This is the meaning of the vertical-pointing arrowheads shown on the trend graph: momentary saturation of the output all the way up to 100% (or down to 0%).

When combined into one PID output, the three actions produce this response:

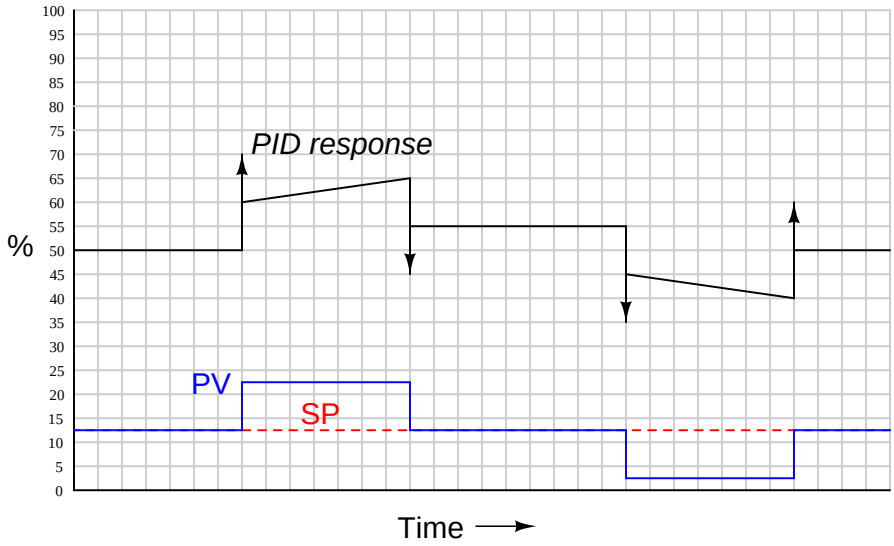


3.1.3 Responses to two momentary steps-and-returns

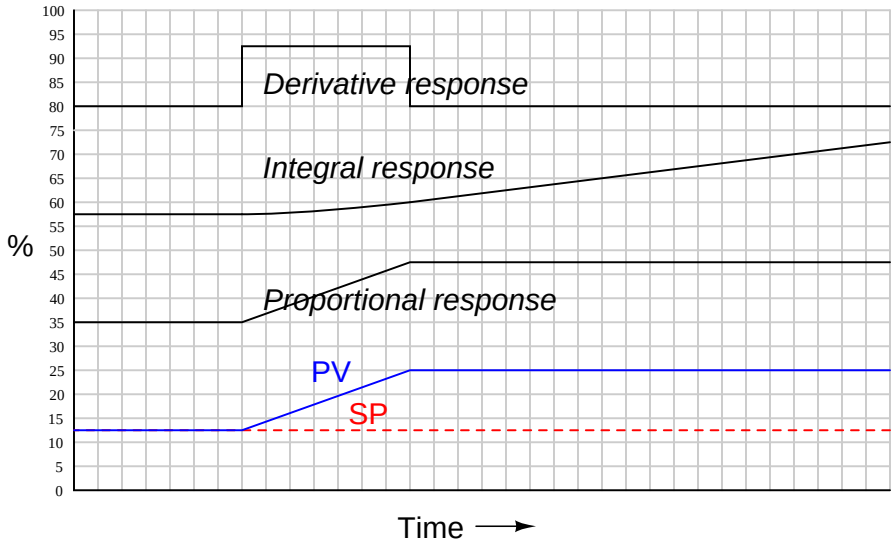


Proportional action directly mimics the shape of all input changes. Integral action ramps at a rate proportional to the magnitude of the input step, for as long as the PV is unequal to the SP. Once PV = SP again, integral action stops ramping and simply holds the last value. Derivative action interprets each step as an *infinite* rate of change, and so generates a "spike" at the leading and at the trailing edges of each step. Note how a leading (rising) edge causes derivative action to saturate high, while a trailing (falling) edge causes it to saturate low.

When combined into one PID output, the three actions produce this response:

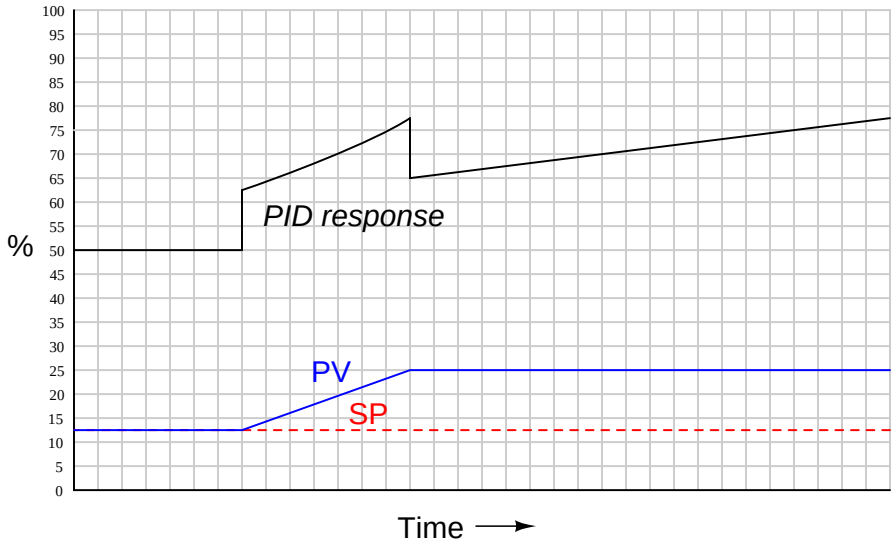


3.1.4 Responses to a ramp-and-hold

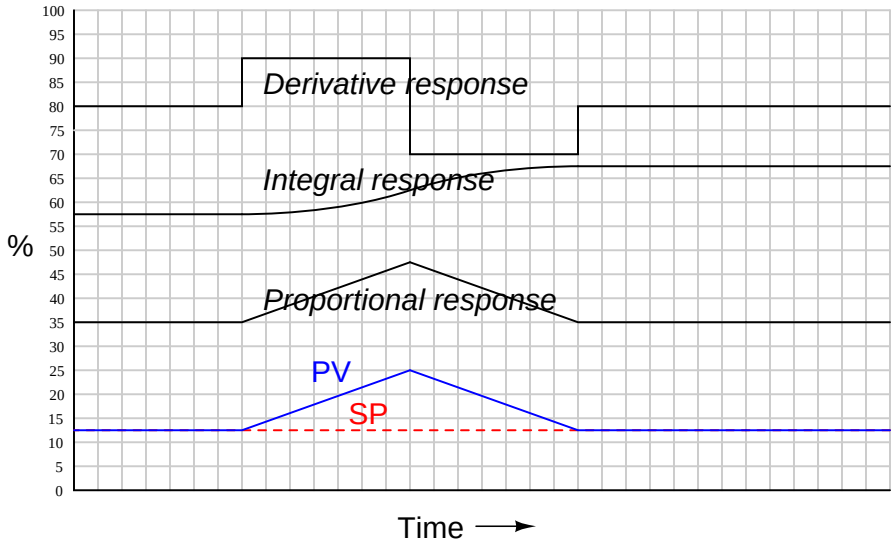


Proportional action directly mimics the ramp-and-hold shape of the input. Integral action ramps slowly at first (when the error is small) but increases ramping rate as error increases. When error stabilizes, integral rate likewise stabilizes. Derivative action offsets the output according to the input's ramping rate.

When combined into one PID output, the three actions produce this response:

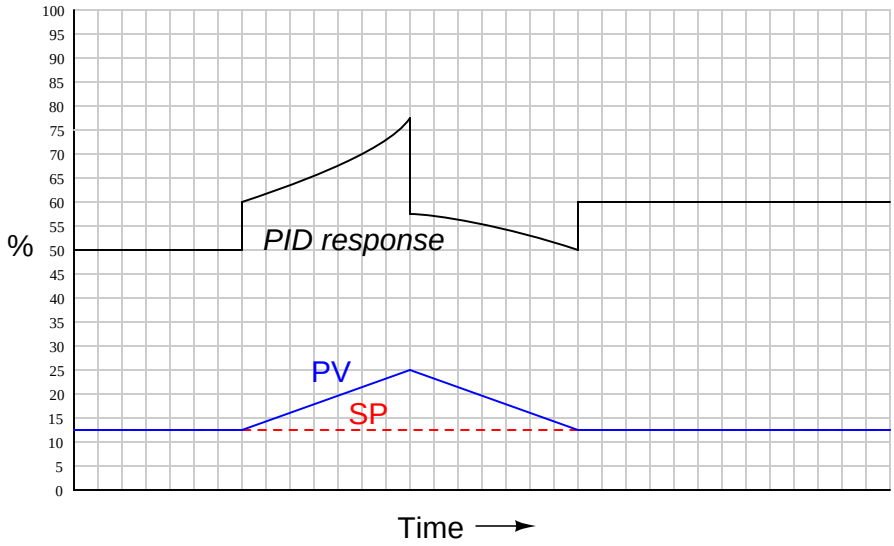


3.1.5 Responses to an up-and-down ramp

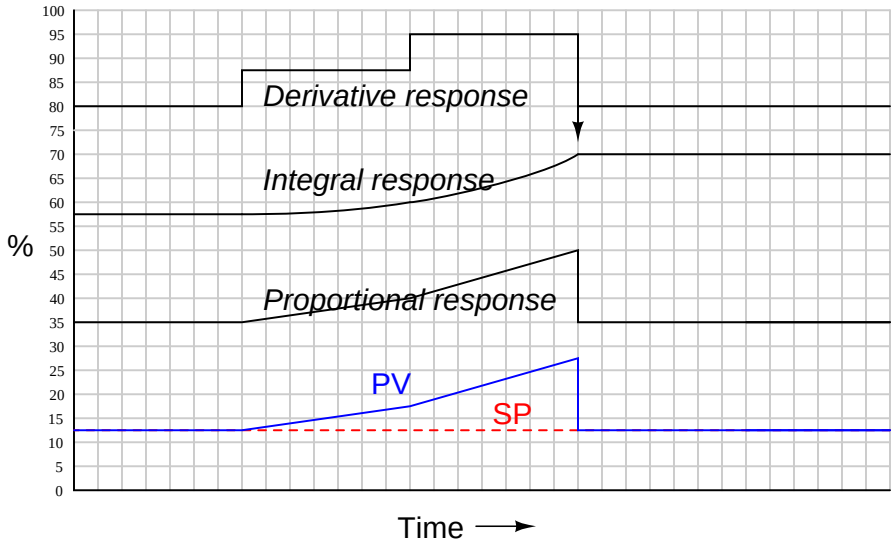


Proportional action directly mimics the up-and-down ramp shape of the input. Integral action ramps slowly at first (when the error is small) but increases ramping rate as error increases, then ramps slower as error decreases back to zero. Once PV = SP again, integral action stops ramping and simply holds the last value. Derivative action offsets the output according to the input's ramping rate: first positive then negative.

When combined into one PID output, the three actions produce this response:

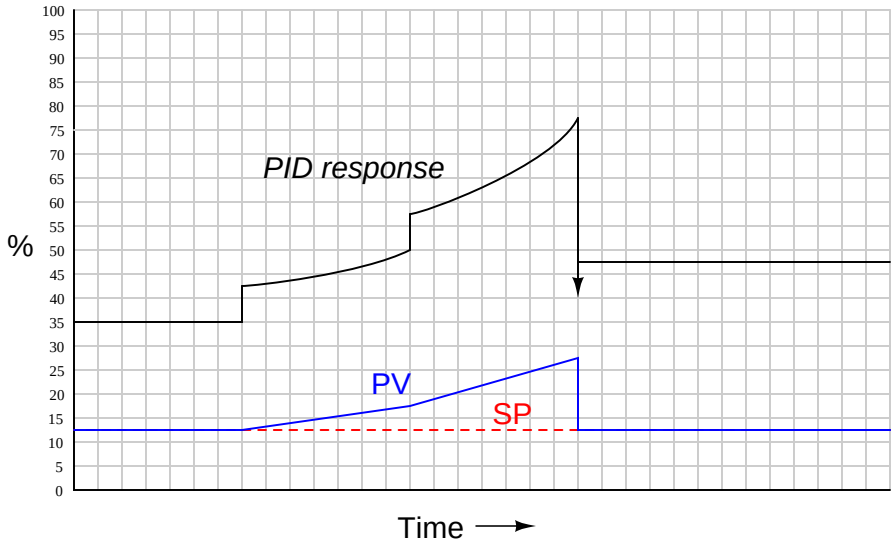


3.1.6 Responses to a multi-slope ramp

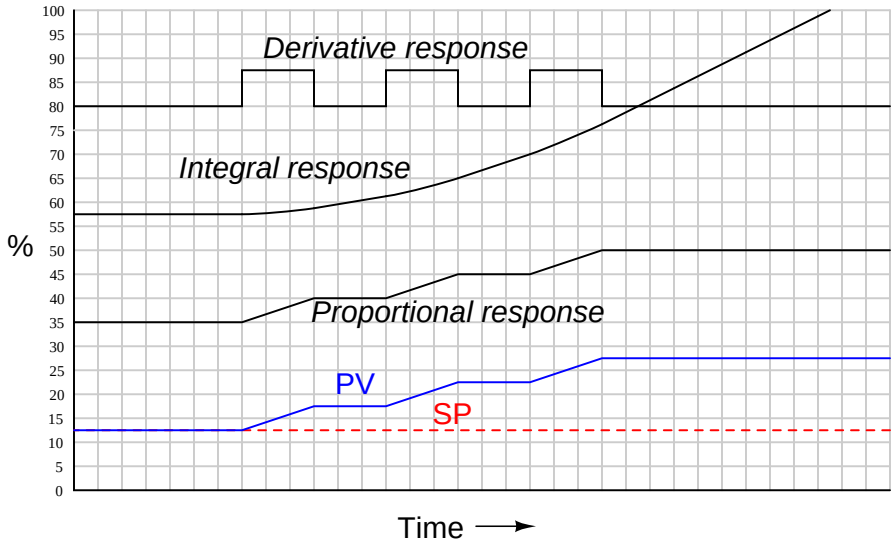


Proportional action directly mimics the ramp shape of the input. Integral action ramps slowly at first (when the error is small) but increases ramping rate as error increases, then accelerates its increase as the PV ramps even steeper. Once PV = SP again, integral action stops ramping and simply holds the last value. Derivative action offsets the output according to the input's ramping rate: first positive, then more positive, then it spikes negative when the PV suddenly returns to SP.

When combined into one PID output, the three actions produce this response:

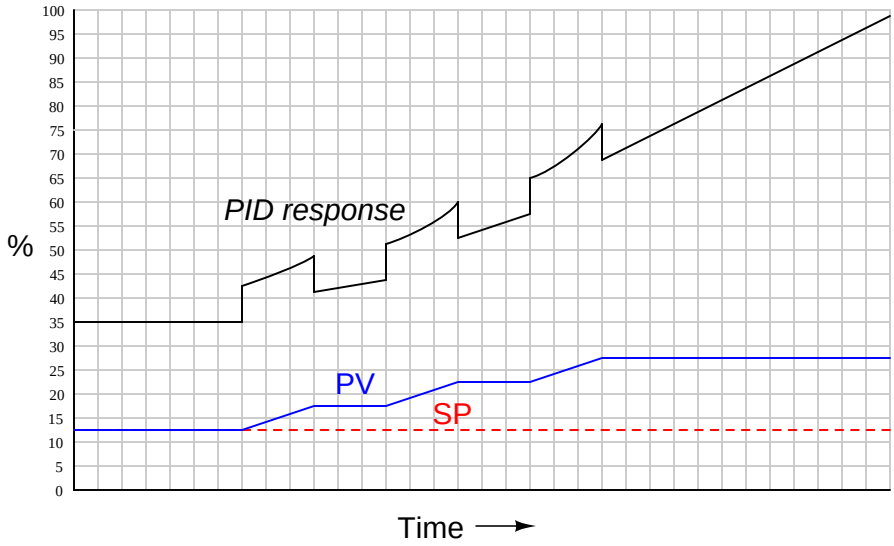


3.1.7 Responses to a multiple ramps and steps

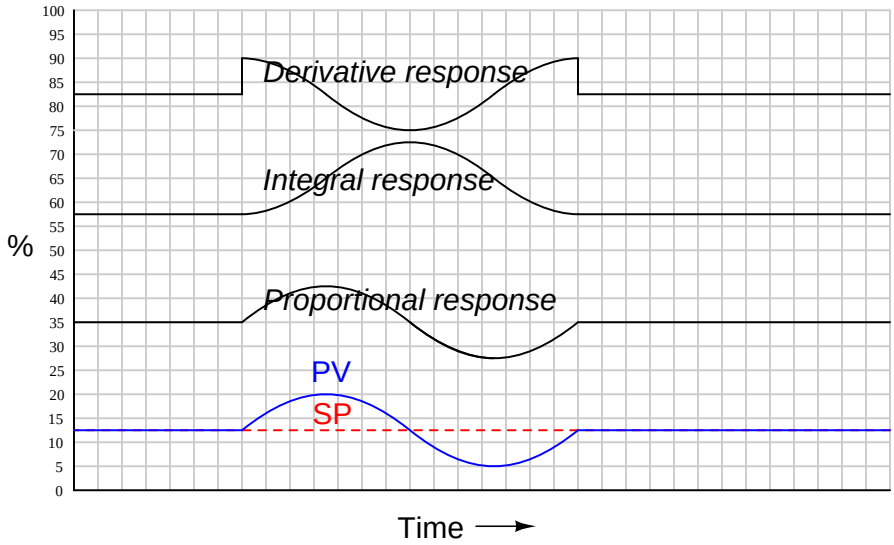


Proportional action directly mimics the ramp-and-step shape of the input. Integral action ramps slowly at first (when the error is small) but increases ramping rate as error increases. Which each higher ramp-and-step in PV, integral action winds up at an ever-increasing rate. Since PV never equals SP again, integral action never stops ramping upward. Derivative action steps with each ramp of the PV.

When combined into one PID output, the three actions produce this response:



3.1.8 Responses to a sine wavelet



As always, proportional action directly mimics the shape of the input. The 90^o phase shift seen in the integral and derivative responses, compared to the PV wavelet, is no accident or coincidence. The derivative of a sinusoidal function is *always* a cosine function, which is mathematically identical to a sine function with the angle advanced by 90^o:

$$\frac{d}{dx}(\sin x) = \cos x = \sin(x + 90^o)$$

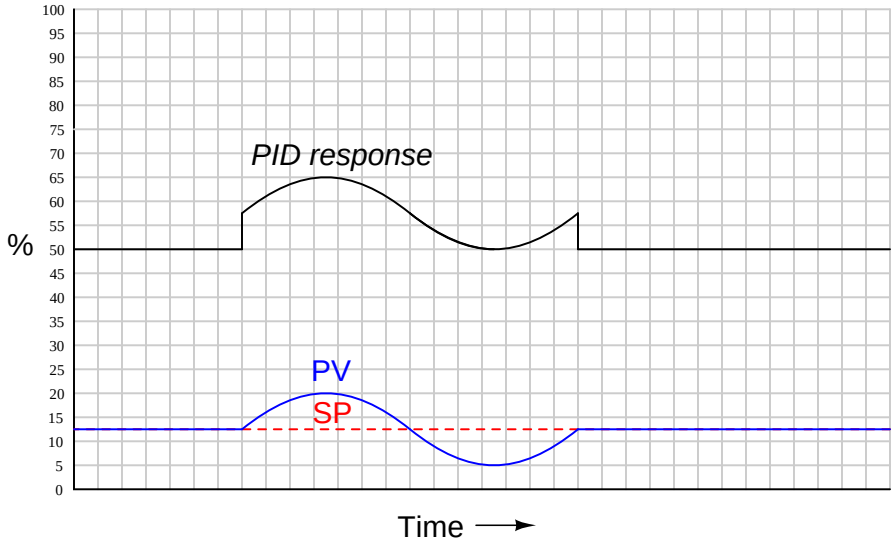
Conversely, the integral of a sine function is *always* a negative cosine function[4], which is mathematically identical to a sine function with the angle retarded by 90^o:

$$\int \sin x \, dx = -\cos x = \sin(x - 90^o)$$

In summary, the derivative operation always adds a positive (leading) phase shift to a sinusoidal input waveform, while the integral operation always adds a negative (lagging) phase shift to a sinusoidal input waveform.

[4] In this example, I have omitted the constant of integration (C) to keep things simple. The actual integral is as such: $\int \sin x \, dx = -\cos x + C = \sin(x - 90^o) + C$. This constant value is essential to explaining why the integral response does not immediately "step" like the derivative response does at the beginning of the PV sine wavelet.

When combined into one PID output, these particular integral and derivative actions mostly cancel, since they happen to be sinusoidal wavelets of equal amplitude and opposite phase. Thus, the only way that the final (PID) output differs from proportional-only action in this particular case is the "steps" caused by derivative action responding to the input's sudden rise at the beginning and end of the wavelet:



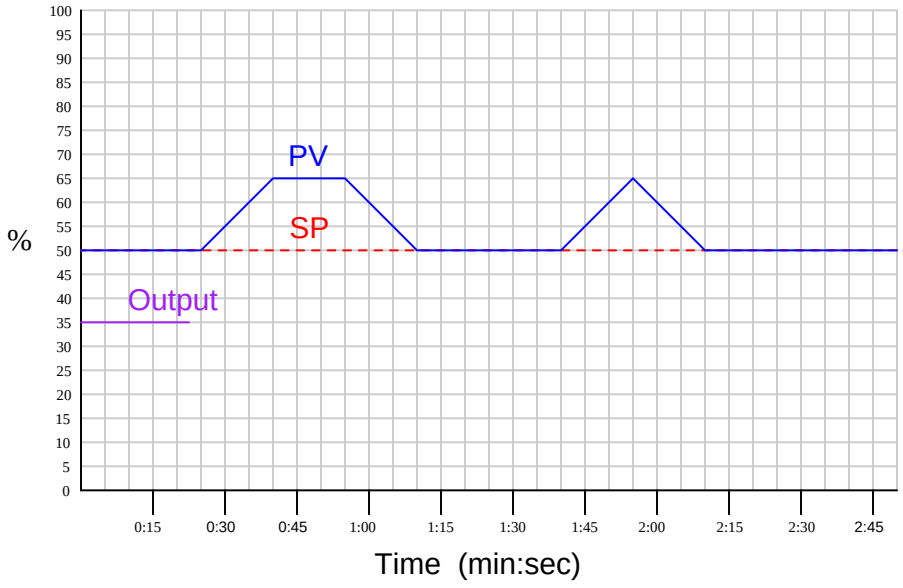
If the I and D tuning parameters were such that the integral and derivative responses were *not* equal in amplitude, their effects would not completely cancel. Rather, the resultant of P, I, and D actions would be a sine wavelet having a phase shift somewhere between -90^o and $+90^o$ exclusive, depending on the relative strengths of the P, I, and D actions.

The 90 degree phase shifts associated with the integral and derivative operations are useful to understand when tuning PID controllers. If one is familiar with these phase shift relationships, it is relatively easy to analyze the response of a PID controller to a sinusoidal input (such as when a process oscillates following a sudden load or setpoint change) to determine if the controller's response is dominated by any one of the three actions. This may be helpful in "de-tuning" an over-tuned (overly aggressive) PID controller, if an excess of P, I, or D action may be identified from a phase comparison of PV and output waveforms.

3.1.9 Note to students regarding quantitative graphing

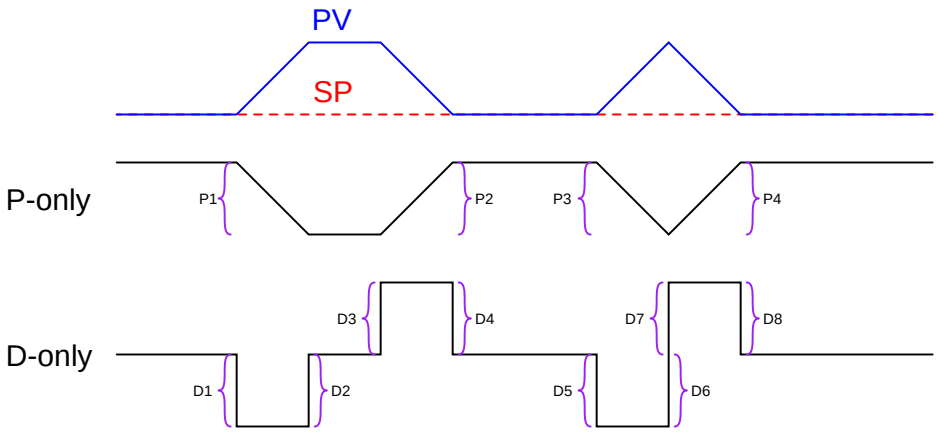
A common exercise for students learning the function of PID controllers is to practice graphing a controller's output given input (PV and SP) conditions, either qualitatively or quantitatively. This can be a frustrating experience for some students, as they struggle to accurately combine the effects of P, I, and/or D responses into a single output trend. Here, I will present a way to ease the pain.

Suppose for example you were tasked with graphing the response of a PD (proportional + derivative) controller to the following PV and SP inputs over time. You are told the controller has a gain of 1, a derivative time constant of 0.3 minutes, and is reverse-acting:

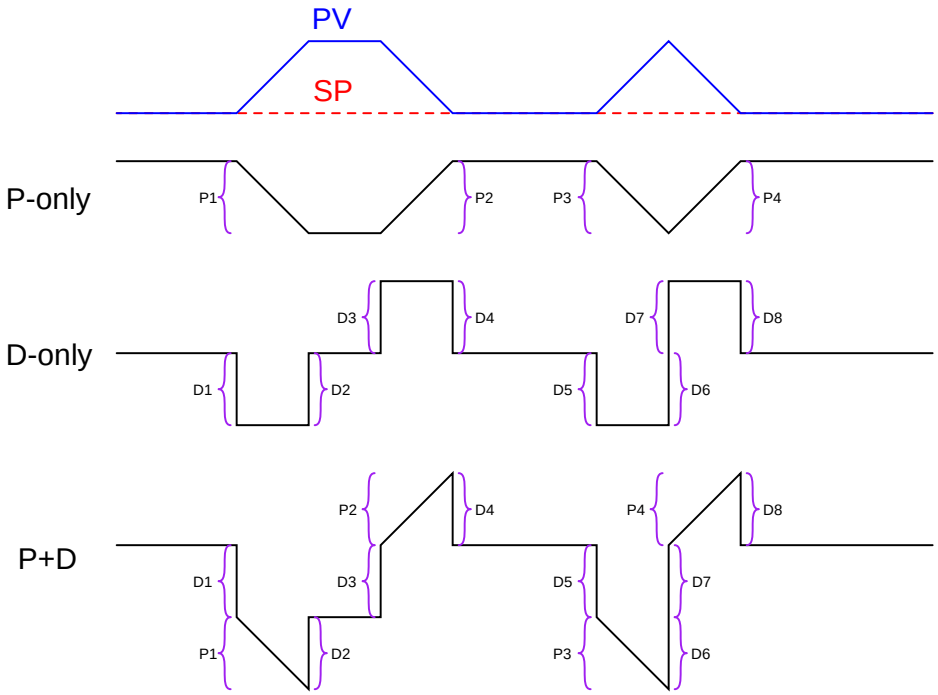


My first recommendation is to *qualitatively* sketch the individual P and D responses. Simply draw two different trends, each one right above or below the given PV/SP trends, showing the shapes of each response over time. You might even find it easier to do if you re-draw the original PV and SP trends on a piece of non-graph paper with the qualitative P and D trends also sketched on the same piece of non-graph paper. The purpose of the qualitative sketches is to separate the task of determining shapes from the task of determining numerical values, in order to simplify the process.

After sketching the separate P and D trends, label each one of the "features" (changes either up or down) in these qualitative trends. This will allow you to more easily combine the effects into one output trend later:



Now, you may qualitatively sketch an output trend combining each of these "features" into one graph. Be sure to label each ramp or step originating with the separate P or D trends, so you know where each "feature" of the combined output graph originates from:

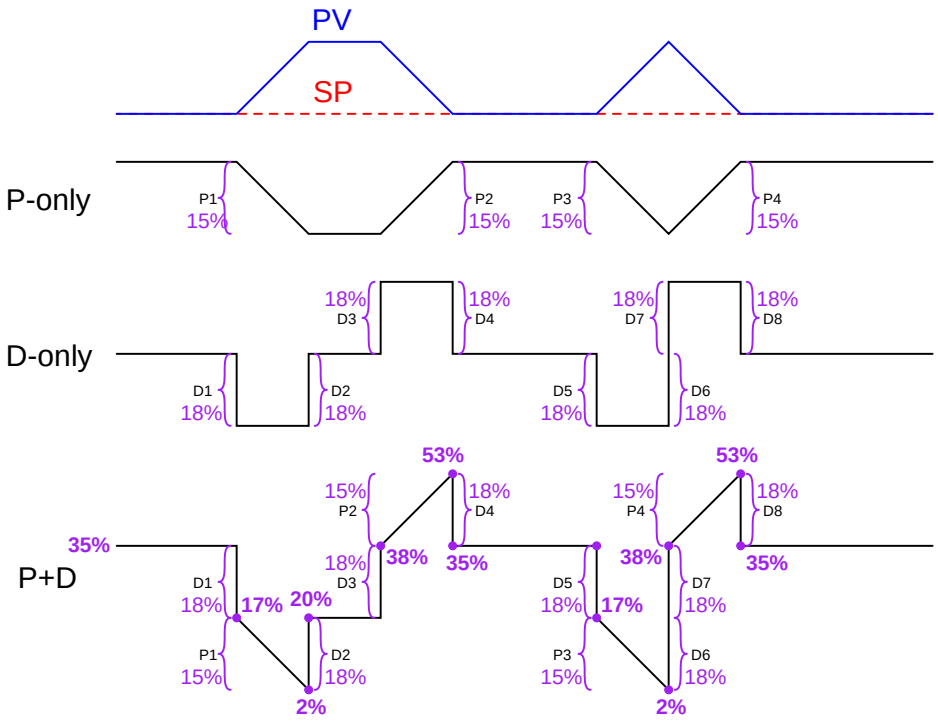


Once the general shape of the output has been qualitatively determined, you may go back to the separate P and D trends to calculate numerical values for each of the labeled "features."

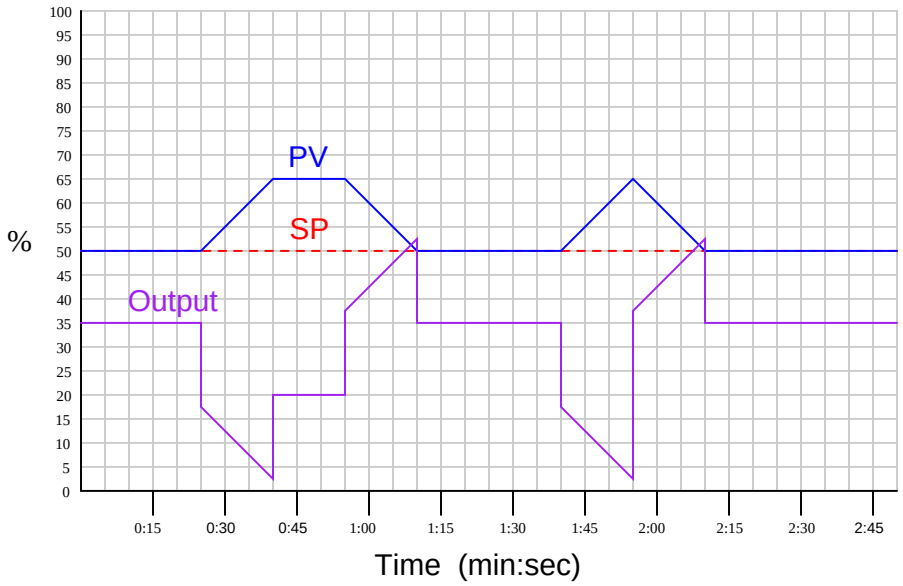
Note that each of the PV ramps is 15% in height, over a time of 15 seconds (one-quarter of a minute). With a controller gain of 1, the proportional response to each of these ramps will also be a ramp that is 15% in height.

Taking our given derivative time constant of 0.3 minutes and multiplying that by the PV's rate-of-change ($\frac{d\text{PV}}{dt}$) during each of its ramping periods (15% per one-quarter minute, or 60% per minute) yields a derivative response of 18% during each of the ramping periods. Thus, each derivative response "step" will be 18% in height.

Going back to the qualitative sketches of P and D actions, and to the combined (qualitative) output sketch, we may apply the calculated values of 15% for each proportional ramp and 18% for each derivative step to the labeled "features." We may also label the starting value of the output trend as given in the original problem (35%), to calculate actual output values at different points in time. Calculating output values at specific points in the graph becomes as easy as cumulatively adding and subtracting the P and D "feature" values to the starting output value:



Now that we know the output values at all the critical points, we may quantitatively sketch the output trend on the original graph:



Chapter 4

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*.

4.1 Programming in C++

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

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

int main (void)
{
  float x, y;

  x = 200;
  y = -560.5;

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

  cout << endl;

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

  return 0;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
./myprogram.exe
```

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

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

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

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

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

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

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

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

4.2 Programming in Python

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
x = 200
y = -560.5

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

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

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

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

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

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

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

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

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

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

4.3 Introduction to pseudocode

Pseudocode is a form of text-based programming intended only for human reading, yet similar enough in syntax and structure to real computer programming languages for a human programmer to be able to easily translate to a high-level programming language such as C++, Python, etc. Since pseudocode is not a formal computer language, we may use it to very efficiently describe certain algorithms (procedures) without having to abide by strict "grammatical" rules as we would if writing in a formal programming language. There is no agreed-upon standard for pseudocode, but here I will outline my own conventions.

4.3.1 Program loops

Each line of text in the following listing represents a command for the digital computer to follow, one by one, in order from top to bottom. The `LOOP` and `ENDLOOP` markers represent the boundaries of a program *loop*, where the same set of encapsulated commands are executed over and over again in cyclic fashion:

Pseudocode listing [7]

```
LOOP
  PRINT "Hello World!"  // This line prints text to the screen
  OUTPUT audible beep on the speaker  // This line beeps the speaker
ENDLOOP
```

In this particular case, the result of this program's execution is a continuous printing of the words "Hello World!" to the computer's display with a single "beep" tone accompanying each printed line. The words following a double-slash (//) are called *comments*, and exist only to provide explanatory text for the human reader, not the computer. Admittedly, this example program would be both impractical and annoying to actually run in a computer, but it does serve to illustrate the basic concept of a program "loop" shown in pseudocode.

[7] I have used a typesetting convention to help make my pseudocode easier for human beings to read: all formal commands appear in bold-faced blue type, while all comments appear in italicized red type. All other text appears as normal-faced black type. One should remember that the computer running any program cares not for how the text is typeset: all it cares is that the commands are properly used (i.e. no "grammatical" or "syntactical" errors).

4.3.2 Assigning values

For another example of pseudocode, consider the following program. This code causes a variable (x) in the computer's memory to alternate between two values of 0 and 2 indefinitely:

Pseudocode listing

```
DECLARE x to be an integer variable
SET x = 2  // Initializing the value of x

LOOP
  // This SET command alternates the value of x with each pass
  SET x = 2 - x
ENDLOOP
```

The first instruction in this listing declares the type of variable x will be. In this case, x will be an *integer* variable, which means it may only represent whole-number quantities and their negative counterparts – no other values (e.g. fractions, decimals) are possible. If we wished to limit the scope of x even further to represent just 0 or 1 (i.e. a single bit), we would have to declare it as a *Boolean* variable. If we required x to be able to represent fractional values as well, we would have to declare it as a *floating-point* variable. Variable declarations are important in computer programming because it instructs the computer how much space in its random-access memory to allocate to each variable, which necessarily limits the range of numbers each variable may represent.

The next instruction initializes x to a value of two. Like the declaration, this instruction need only happen once at the beginning of the program's execution, and never again so long as the program continues to run. The single `SET` statement located between the `LOOP` and `ENDLOOP` markers, however, repeatedly executes as fast as the computer's processor allows, causing x to rapidly alternate between the values of two and zero.

It should be noted that the "equals" sign (=) in computer programming often has a different meaning from that commonly implied in ordinary mathematics. When used in conjunction with the `SET` command, an "equals" sign *assigns* the value of the right-hand quantity to the left-hand variable. For example, the command `SET x = 2 - x` tells the computer to first calculate the quantity $2 - x$ and then set the variable x to this new value. It definitely does *not* mean to imply x is actually equal in value to $2 - x$, which would be a mathematical contradiction. Thus, you should interpret the `SET` command to mean "set equal to . . ."

4.3.3 Testing values (conditional statements)

If we mean to simply test for an equality between two quantities, we may use the same symbol (=) in the context of a different command, such as "IF":

Pseudocode listing

```
DECLARE x to be an integer variable

LOOP

  // (other code manipulating the value of x goes here)

  IF x = 5 THEN
    PRINT "The value of the number is 5"
    OUTPUT audible beep on the speaker
  ENDIF
ENDLOOP
```

This program repeatedly tests whether or not the variable x is equal to 5, printing a line of text and producing a "beep" on the computer's speaker if that test evaluates as true. Here, the context of the IF command tells us the equals sign is a test for equality rather than a command to assign a new value to x. If the condition is met ($x = 5$) then all commands contained within the IF/ENDIF set are executed.

Some programming languages draw a more explicit distinction between the operations of equality test versus assignment by using different symbol combinations. In C and C++, for example, a single equals sign (=) represents assignment while a double set of equals signs (==) represents a test for equality. In Structured Text (ST) PLC programming, a single equals sign (=) represents a test for equality, while a colon plus equals sign (:=) represents assignment. The combination of an exclamation point and an equals sign (!=) represents "not equal to," used as a test condition to check for *inequality* between two quantities.

4.3.4 Branching and functions

A very important feature of any programming language is the ability for the path of execution to change (i.e. the program "flow" to *branch* in another direction) rather than take the exact same path every time. We saw shades of this with the `IF` statement in our previous example program: the computer would print some text and output a beep sound if the variable x happened to be equal to 5, but would completely skip the `PRINT` and `OUTPUT` commands if x happened to be any other value.

An elegant way to modularize a program into separate pieces involves writing portions of the program as separate *functions* which may be "called" as needed by the main program. Let us examine how to apply this concept to the following conditional program:

Pseudocode listing

```
DECLARE x to be an integer variable

LOOP

  // (other code manipulating the value of x goes here)

  IF x = 5 THEN
    PRINT "The value of the number is 5"
    OUTPUT audible beep on the speaker
  ELSEIF x = 7 THEN
    PRINT "The value of the number is 7"
    OUTPUT audible beep on the speaker
  ELSEIF x = 11 THEN
    PRINT "The value of the number is 11"
    OUTPUT audible beep on the speaker
  ENDIF
ENDLOOP
```

This program takes action (printing and outputting beeps) if ever the variable x equals either 5, 7, or 11, but not for any other values of x. The actions taken with each condition are quite similar: print the numerical value of x and output a single beep. In fact, one might argue this code is ugly because we have to keep repeating one of the commands verbatim: the `OUTPUT` command for each condition where we wish to computer to output a beep sound.

We may streamline this program by placing the PRINT and OUTPUT commands into their own separate "function" written outside the main loop, and then *call* that function whenever we need it. The boundaries of this function's code are marked by the BEGIN and END labels shown near the bottom of the listing:

Pseudocode listing

```
DECLARE n to be an integer variable
DECLARE x to be an integer variable
DECLARE PrintAndBeep to be a function

LOOP

  // (other code manipulating the value of x goes here)

  IF x = 5 OR x = 7 OR x = 11 THEN
    CALL PrintAndBeep(x)
  ENDIF
ENDLOOP


BEGIN PrintAndBeep (n)
  PRINT "The value of the number is" (n) "!"
  OUTPUT audible beep on the speaker
  RETURN
END PrintAndBeep
```

The main program loop is much shorter than before because the repetitive tasks of printing the value of x and outputting beep sounds has been moved to a separate function. In older computer languages, this was known as a *subroutine*, the concept being that flow through the main program (the "routine") would branch to a separate sub-program (a "subroutine") to do some specialized task and then return back to the main program when the sub-program was done with its task.

Note that the program execution flow never reaches the PrintAndBeep function unless x happens to equal 5, 7, or 11. If the value of x never matches any of those specific conditions, the program simply keeps looping between the LOOP and ENDLOOP markers.

Note also how the value of x gets *passed* on to the PrintAndBeep function, then read inside that function under another variable name, n. This was not strictly necessary for the purpose of printing the value of x, since x is the only variable in the main program. However, the use of a separate ("local") variable within the PrintAndBeep function enables us at some later date to use that function to act on other variables within the main program while avoiding conflict. Take this program for example:

Pseudocode listing

```
DECLARE n to be an integer variable
DECLARE x to be an integer variable
DECLARE y to be an integer variable
DECLARE PrintAndBeep to be a function

LOOP

  // (other code manipulating the value of x and y goes here)

  IF x = 5 OR x = 7 OR x = 11 THEN
    CALL PrintAndBeep(x)
  ENDIF
  IF y = 0 OR y = 2 THEN
    CALL PrintAndBeep(y)
  ENDIF
ENDLOOP


BEGIN PrintAndBeep (n)
  PRINT "The value of the number is" (n) "!"
  OUTPUT audible beep on the speaker
  RETURN
END PrintAndBeep
```

Here, the PrintAndBeep function gets used to print certain values of x, then re-used to print certain values of y. If we had used x within the PrintAndBeep function instead of its own variable (n), the function would only be useful for printing the value of x. Being able to pass values to functions makes those functions more useful.

A final note on branching and functions: most computer languages *allow a function to call itself* if necessary! This concept is known as *recursion* in computer science.

Chapter 5

Questions

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

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

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

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

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

General challenges following tutorial reading

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

General follow-up challenges for assigned problems

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

General follow-up challenges for experiments or projects

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

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

5.1 Conceptual reasoning

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

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

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

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

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

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

5.1.1 Reading outline and reflections

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

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

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

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

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

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

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

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

5.1.2 Foundational concepts

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

Energy

Conservation of Energy

Simplification as a problem-solving strategy

Thought experiments as a problem-solving strategy

Limiting cases as a problem-solving strategy

Annotating diagrams as a problem-solving strategy

Interpreting intermediate results as a problem-solving strategy

Graphing as a problem-solving strategy

Converting a qualitative problem into a quantitative problem

Converting a quantitative problem into a qualitative problem

Working "backwards" to validate calculated results

Reductio ad absurdum

Re-drawing schematics as a problem-solving strategy

Cut-and-try problem-solving strategy

Algebraic substitution

???

5.1.3 First conceptual question

Challenges

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

5.1.4 Second conceptual question

Challenges

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

5.1.5 Applying foundational concepts to ???

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

(Under development)

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

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

Challenges

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

5.1.6 Explaining the meaning of calculations

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

Schematic diagram of the ??? circuit:

(Under development)

Calculations performed in order from first to last:

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

Challenges

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

5.1.7 Explaining the meaning of code

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

Schematic diagram of the ??? circuit:

(Under development)

Code listing:

```
#include <stdio.h>

int main (void)
{

  return 0;
}
```

Challenges

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

5.2 Quantitative reasoning

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

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

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

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

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

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

5.2.1 Miscellaneous physical constants

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.2 Introduction to spreadsheets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.3 First quantitative problem

Challenges

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

5.2.4 Second quantitative problem

Challenges

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

5.2.5 ??? simulation program

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

Challenges

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

5.3 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.

5.3.1 First diagnostic scenario

Challenges

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

5.3.2 Second diagnostic scenario

Challenges

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

Chapter 6

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!*

6.1 Recommended practices

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

6.1.1 Safety first!

Electricity, when passed through the human body, causes uncomfortable sensations and in large enough measures[1] 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 contact*[2] *with an energized conductor, no matter what the circuit's voltage*[3] *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

[1] Professor 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.

[2] By "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.

[3] Another 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.

6.1.2 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 kΩ or greater than 100 kΩ, unless such values are definitely necessary[4]. Resistances below 1 kΩ 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 kΩ 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 kΩ and 100 kΩ, 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 breadboards[5], 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.

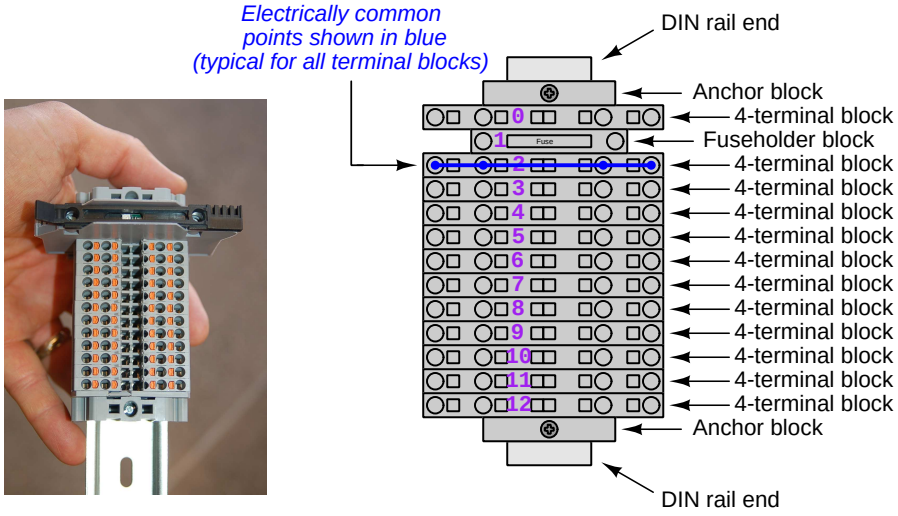
[4] An example of a necessary resistor value much less than 1 kΩ is a *shunt resistor* used to produce a small voltage drop for the purpose of sensing current in a circuit. Such shunt resistors must be low-value in order not to impose an undue load on the rest of the circuit. An example of a necessary resistor value much greater than 100 kΩ 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).

[5] Admittedly, 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.

6.1.3 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 breadboards[6]. 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 rail[7] 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 other[8] 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 buttons[9] for this task which may be pressed using the tip of any suitable tool.

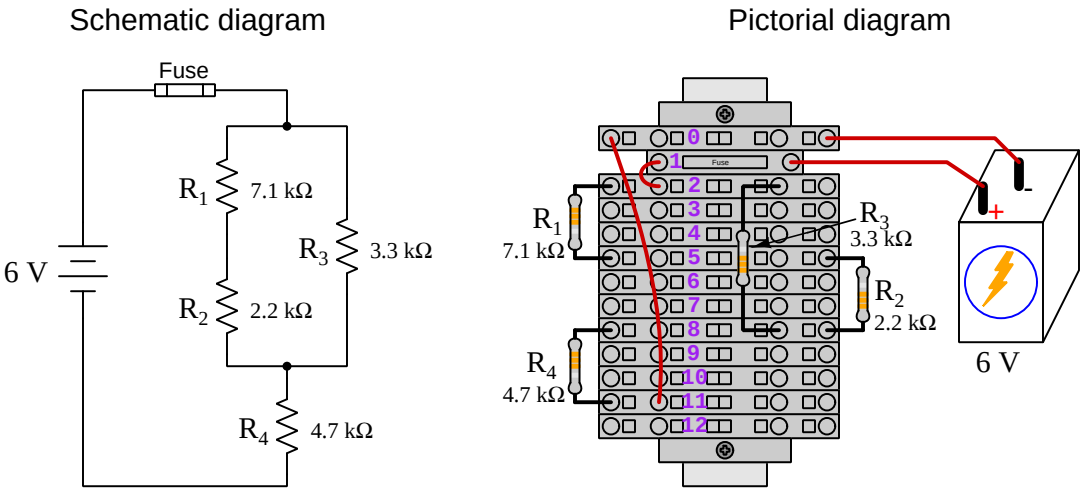
[6] Solderless breadboard are preferable for complicated electronic circuits with multiple integrated "chip" components, but for simpler circuits I find terminal blocks much more practical. An alternative to solderless breadboards for "chip" circuits is to solder chip sockets onto a PCB and then use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

[7] DIN 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.

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

[9] The 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 SPICE[10] netlists, where component connections are identified by terminal number:

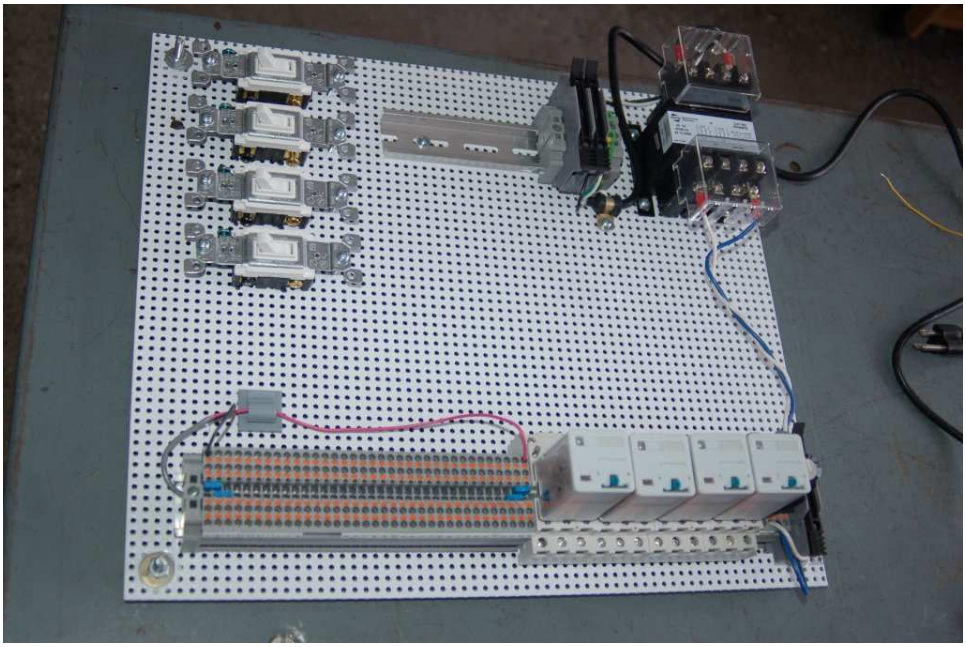
```
* Series-parallel resistor circuit
v1 1 0 dc 6
r1 2 5 7100
r2 5 8 2200
r3 2 8 3300
r4 8 11 4700
rjmp1 1 2 0.01
rjmp2 0 11 0.01
.op
.end
```

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.

[10] SPICE 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 subpanel[11]. 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 with a diameter suitable for tapping with 6-32 machine screw threads, their presence making it very easy to attach other sections of DIN rail, printed circuit boards, or specialized electrical components directly to the grounded metal subpanel. Such a "terminal block board" is an inexpensive[12] yet highly flexible means to construct physically robust circuits using industrial wiring practices.

[11] An 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.

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

6.1.4 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 testable[14], 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 available[15].

[13] Science 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 scient*ists* 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.

[14] This is the principle of *falsifiability*: that a scientific statement has value only insofar as it is liable to disproof 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.

[15] A 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.

6.1.5 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.

6.2 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!
- ???.
- ???.

6.3 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 environment[1] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[2] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[3] 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

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

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

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

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

Another 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 varied[4] 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 challenge[5] one another.

To high standards of education,

Tony R. Kuphaldt

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

[5] Avoid 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 bread.

`SPICE` circuit simulation program

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

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

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

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

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

Appendix E

References

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

18 February 2025 – formatting and other minor edits made to the Tutorial chapter. Also added a new section to the Tutorial introducing readers to common process-control terminology.

2-3 February 2025 – document first created.

Index