Looping is a powerful programming technique that allows Matlab users to execute a set of code multiple times, or over every value in a vector. For example, if you need to run a simulation in which a filter’s response to variations in resistor tolerance is calculated and displayed for 1,000 random perturbations in resistances, or calculate the output of a circuit whose input is a complicated input waveform sampled at 1,000 different times, the for-end loop makes such applications possible.

**OBJECTIVES**

After completing this chapter, you will be able to use Matlab to do the following:

- Program using **for-end** loops
- Understand how to index vectors within for loops
- Nest loops inside each other
- Create functions that call other user functions
- Program using **while-end** loops
- Understand how to preallocate memory to speed program execution


**disp()**

Previously, `fprintf()` was introduced as a function to print text and numbers to the screen, possibly using formatting characters including tabs and newlines. A simpler command that can print either pure string text or a variable containing a number or vector, but not combined text with numbers, is `disp()`. For example:

```matlab
disp('Hello world')
v = [1 2 3 4]; disp(v)
```

will cause the following to print to the command window:

```
Hello world
1 2 3 4
```

Use `disp()` instead of `fprintf()` when you only need to print a simple variable. It’s frequently helpful when debugging to put in a `disp()` statement to check how the code is evaluating inside a function.

**for-end loop**

The **for-end** loop is used to repeat a section of code for a known number of times, as in the example below:

```matlab
for i=1:10
    disp('ECE is fun');
end
```

The **for-end** loop requires the creation of a loop variable, here called `i`, and a loop vector, here the set of numbers from 1 to 10. The loop executes as many times as there are entries in the loop vector, and for each iteration the loop variable takes on a successive value of the loop vector. There can be any number of command lines inside the **for-end** loop. Notice how these commands are typically indented to make them easier to read, much like the commands inside the **if-else-end** block are indented.
For example:

    for i=1:5
        disp(i)
    end

will print the following to the command window:

    1
    2
    3
    4
    5

The following code makes a list of the squares and cubes of the even numbers:

    for x=2:2:6
        disp([x  x^2  x^3])
    end

The above code prints the following to the command line:

    2     4     8
    4    16    64
    6    36   216

**PRACTICE PROBLEMS**

1. What does the following function do? Try to figure it out before you run it to check. Pass in a value between 5 and 10 to check your understanding.

    function classproblem1(n)
    x=1;
    for i=1:n
        x = x+x;
        disp(x)
    end
    end
2. Fill in the missing part below, labeled ***, to create a function that takes a single resistor value r, returns nothing, and displays the value of r in parallel with resistors of standard values 10, 15, 22, 33, 47, 68, 82 using a for loop. Test it with a value of 30. ©®

```
function classproblem2(r)
    for r2=[10 15 22 33 47 68 82];
        rp = (*** a function of r and r2 *** );
        disp(rp)
    end
end
```

3. Write a function that takes no arguments, returns no arguments, and prints a two-column list that converts degrees to radians. It should print from 0 to 180 degrees in 5 degree increments. ©

---

**FOR-END LOOPS INDEXING VECTORS**

One of the most common uses of a for loop is to index a vector. This permits complex operations to be applied to vectors. For example, a different way to accomplish the previous class problem is in the following function:

```
function ForIndexingVectorExample(r)
    vAllResistors = [10 15 22 33 47 68 82];
    for i = 1:length(vAllResistors)
        rCur = vAllResistors(i);
        rp = 1/(1/r+1/rCur);
        disp(rp)
    end
end
```

This example begins by defining a vector holding all standard values resistors called vAllResistors. Then it creates a loop variable i which iterates from 1 to the
number of resistors in \( \text{vAllResistors} \). For each iteration, it pulls one resistor out of the vector and calls it \( \text{rCur} \). Then it finds the parallel combination of \( \text{rCur} \) with whatever resistor value was passed into the function and displays the result.

Up to this point, vector math has been handled with a single operation, for example multiplication of vector \( \text{v} \) by a constant \( c \) is the single operation: \( \text{v} \times c \). This operates on the entire vector at once and is very efficient, but some operations are more easily understood using a \textbf{for} loop that steps through every value inside the vector. For instance, consider a piecewise-linear function called the step function and abbreviated \( \text{u(t)} \), defined and graphed as follows:

\[
\text{u(t)} = \begin{cases} 
0, & t < 0 \\
1, & t \geq 0 
\end{cases}
\]

A function that plots this same function is given below:

```matlab
function plotStepFunction() % takes and returns
% no arguments
vt = linspace(-2, 3, 1000); % horizontal axis
% has 1000 points
vu = zeros(1,1000); % create vertical data vector
% of same size
for i=1:100         %
t = vt(i); % t loops through each point
% in vector vt
```
if (t<0) % if t is negative
    vu(i) = 0; % then vu at that time is 0
else % otherwise
    vu(i) = 1; % vu at that time is 1
end
end
plot(vt,vu)

Notice how the for-end loop sets t to every possible value of the vt vector, and then individually performs some operation on it – here, checking to see if it is positive or negative, and then creates the output vector accordingly.

The more complex example plot below plots the following mathematical function from 0 ≤ t ≤ 30:

\[
f(t) = \begin{cases} 
\sqrt{10t}, & 0 \leq t \leq 10 \\
10e^{10-t}, & 10 \leq t \leq 20 \\
t - 20, & t \geq 20 
\end{cases}
\]

function plotPiecewiseFunction()
    vt = linspace(0,30,1000); % let t range from 0 to 30
    vf = zeros(1,1000); % create the f vector filled with 0’s
    for i=1:1000
        t = vt(i); % iterate so t will be % every value of vt
        if (t >= 0) && (t < 10)
            vf(i) = sqrt(10*t);
        elseif (t >= 10) && (t <20)
            vf(i) = 10*exp(10-t);
        end
    end
    plot(vt,vf)
end
elseif \( t \geq 20 \)
    \[ v_f(i) = t - 20; \]
end
end
plot(vt,vf)

PRACTICE PROBLEMS

4. Create a function that plots the following for \(-5 \leq t \leq 5\).
   Plot using 100 points.®

   \[
   v(t) = \begin{cases} 
   1 - t, & t < 0 \\
   e^{t/4}, & t \geq 0 
   \end{cases}
   \]
TECH TIP: MONTE CARLO SIMULATIONS

Monte Carlo simulations are commonly used in EE to test how engineering designs respond to random changes in component tolerance. For example, consider the lowpass circuit shown below, which passes low frequencies but tends to block frequencies above $f = \frac{1}{2\pi RC}$:

Choose $R = 1k\Omega$ and $C = 1\mu F$ to make the filter pass frequencies below 160Hz. How will this cutoff frequency vary given a resistor tolerance of ±5% and capacitor tolerance of ±10%? A graduate-level course in stochastic theory will derive the result exactly, but it is simpler to simulate the results several thousand times, each time with different random resistor and capacitor values.

In Matlab, `rand` returns a random number between [0,1]. To change this to [-0.05 +0.05] to simulate a 5% resistor tolerance, note there is a desired span of 0.1 in [-0.05 0.05], so $(\text{rand} * 0.1)$ gives the correct span of [0 0.1]. Subtract 0.05 to create the desired range [-0.05 0.05]. Therefore $\text{rand} * 0.1 - 0.05$ gives a random number between -0.05 and 0.05 for the resistors. Similarly, $\text{rand} * 0.2 - 0.1$ gives a random number from 0.1 to 0.1 for the capacitors.

```matlab
function result = MonteCarlo(N)
% N is the number of simulations desired
result = zeros(1,N); % fill result with N zeros
for i=1:N % do the simulation N times

Computer Tools for Electrical Engineers
R = 1000 + 1000*(rand*0.1-0.05);
% +/-5% resistor tolerance
C = 1e-6 + 1e-6*(rand*0.2-0.1);
% +/-10% capacitor tolerance
f = 1/(2*pi*R*C);
result(i) = f;
end

Run the function from the command window and have it do
1,000 simulations. The results show that most of the critical
frequencies span from 140Hz to 180Hz, something difficult
to derive using methods other than simulation.
result = MonteCarlo(1000);
plot(result,'.')
xlabel('simulation number')
ylabel('critical frequency')

A new Matlab command that can help visualize the results
of Monte Carlo simulations is hist(), short for histogram. By
default it creates ten evenly-spaced bins for the data, counts
how many fall into each bin, and returns the bin centers in a
separate vector. Used in conjunction with `bar()` it plots a histogram of the results.

```
[n,x]=hist(result);  \% n is the number
                      \% in each bin
                      \% centered at x
bar(x,n);  \% plots the resulting histogram
xlabel('critical frequency')
ylabel('number')
```

Now it is apparent that although the average value of the filter’s cutoff is about 160Hz, which is what would be obtained with resistors and capacitors whose values are precisely as marked, using real-world resistors and capacitors with about a 5% and 10% tolerance respectively will result in a spread of cutoff frequencies. One could expect to find cutoffs as low as 140Hz and as high as 180Hz when using components with real-world, imperfect values. Notice how the `hist` command displays the same information as the previous MonteCarlo function does, but in a more easily-interpreted fashion.
PRACTICE PROBLEMS

5. Solve the Monte Carlo filter problem on the previous page, but now assume that the capacitors are higher quality 5% versions, and the resistors have a 1% tolerance. Plot the results of 1,000 simulations as a histogram.

NESTED LOOPS

For-end loops are often nested inside each other. For example, if a parts cabinet were stocked with exactly 4 types of resistors, 10Ω, 22Ω, 47Ω, 51Ω, find all possible combinations of these types:

```matlab
function FindResistors()
    % FindResistors finds all combos of 2 resistors given 4 types
    vr = [10 22 47 51]; % 4 types of stocked resistors
    for i1 = 1:4 % the first resistor index
        R1 = vr(i1); % R1 is the first resistor value
        for i2 = 1:4 % choose the second resistor index
            R2 = vr(i2); % R2 is the second resistor value
            fprintf('R1 = %g, R2 = %g\n', R1, R2)
        end
    end
end
```

This code returns the following output:

- R1 = 10, R2 = 10
- R1 = 10, R2 = 22
- R1 = 10, R2 = 47
- R1 = 10, R2 = 51
- R1 = 22, R2 = 10
R1 = 22, R2 = 22
R1 = 22, R2 = 47
R1 = 22, R2 = 51
R1 = 47, R2 = 10
R1 = 47, R2 = 22
R1 = 47, R2 = 47
R1 = 47, R2 = 51
R1 = 51, R2 = 10
R1 = 51, R2 = 22
R1 = 51, R2 = 47
R1 = 51, R2 = 51

PRACTICE PROBLEMS

6. Modify the above code to print all possible values of R1 and R2 in parallel.

7. Is it possible to sort the above result with no code changes other than by simply inserting the sort() command in the correct place? If so, do it and report the result. If not, explain why it is not possible. This problem should clarify when it is helpful to print results to the screen and when it is helpful to return results as a function's output argument.
USING NESTED LOOPS TO SEARCH FOR EXACT SOLUTIONS

Consider a program that searches for integer solutions for the sides of a right triangle, known as Pythagorean triples, satisfying \( c = \sqrt{a^2 + b^2} \). The program searches for integer hypotenuses as side \( a \) ranges from 1 to some given upper bound \( N \) and as side \( b \) ranges from 1 to \( N \).

Every time an integer hypotenuse \( c \) is found, it is printed to the command window, as shown below:

```matlab
function pythagorean(N)
    % pythagorean(N) searches for integer sides of right
    % triangles for lengths of sides varying from 1 to N
    for a = 1:N % side a
        for b = 1:N % side b
            c = sqrt(a*a+b*b); % side c, the hypotenuse
            if (c==floor(c)) % a solution is found
                disp(c)
            end
        end % loop through side b values
    end % loop through side a values
end
```

**RECALL**

X is an integer only if \( x = \text{floor}(x) \)

**PRACTICE PROBLEMS**

8. The above code for \texttt{pythagorean()} works, but it only prints the hypotenuse values. Modify the code to print all the sides \( a \), \( b \), and \( c \), and list the results for sides \( a \) and \( b \) between 1 and 20.
PRO TIP: NON-ENGINEERING CAREERS

Many electrical engineering graduates choose to pursue careers other than Electrical Engineering, and find that the quantitative skills they learned as students transfer well. Popular non-engineering careers for EE graduates include:

- **Sales:** There are many opportunities for qualified engineers who are more interested in working with people than design. Although these jobs are really about building trusting relationships, many require engineering-level knowledge about the system being sold, such as when selling substation transformers to provide power to growing communities.

- **Project management:** This is another people-centric position, and it requires the ability to manage teams of engineers during the development phase of a project. Such positions require far more communication and planning skills than design skills.

- **Military:** Increasing automation, especially in the technology-centric branches of the Air Force and Navy, calls for greater numbers of engineers. There are often special ROTC scholarships reserved specifically for engineering majors to help fill this gap.

- **Medicine:** Engineers have higher Medical College Admission Test (MCAT) scores on average than both biology majors and “pre-med” health science majors\(^1\), and the analytical training they receive gives them advantages in certain specialties, including cardiology, neurology, and radiology.
Law: Math-intensive undergraduate curricula are correlated with Law School Admission Test (LSAT) performance; according to 2016 data, engineering students are the fifth-highest performing, on average, with physics majors ranking first\(^2\). Patent law recruits heavily from engineering majors; an undergraduate degree in engineering alone is sufficient qualification to take the U.S. Patent Bar Exam.

Opportunities abound both within and outside the engineering profession. Which will you choose?


**USING NESTED LOOPS TO SEARCH FOR BEST SOLUTIONS**

Finding exact solutions, as was done in the previous section, is done by testing with the `==` operator. Sometimes exact solutions do not exist, in which case one must settle for finding the best solution. Finding this best solution requires keeping track of the current best solution, and checking through every iteration to see if the current computation is better. Error is often computed using code of the form `error = abs(currentComputation - desired)` since it is the absolute error that needs to be minimized.
For example, if you need to find the value of two standard value resistors from 1Ω to 1MΩ in series, whose sum is as close as possible to a given desired value, enter the following:

```matlab
function [R1best, R2best] = findclosestseries(rDesired)
% create vR with standard value resistors from 1 to 9.1
vR = [1 1.1 1.2 1.3 1.5 1.6 1.8 2 2.2 2.4 2.7 3 ...
     3.3 3.6 3.9 4.3 4.7 5.1 5.6 6.2 6.8 7.5 8.2 9.1];
% ... wraps the line

% duplicate vR by multiples of 10 to create all
% standard value Rs
vR = [0 vR vR*10 vR*100 vR*1e3 vR*1e4 vR*1e5 1e6];
R1best = 0; % we haven’t yet found the best value of R1
R2best = 0; % we haven’t yet found the best value of R2
bestError = 1e12; % our initial error is huge

% loop through to find every possible value of R1
for i1 = 1:length(vR)
    R1 = vR(i1);
    % loop through to find every possible value of R2
    for i2 = 1:length(vR)
        R2 = vR(i2);
        Rseries = R1+R2;
        currentError = abs(Rseries - rDesired);
        if (currentError < bestError)
            % we found a better solution
            R1best = R1;
            R2best = R2;
            bestError = currentError;
        end
    end
end
```
9. Change the above program to print out the closest value of two standard-value resistors in parallel to a given desired resistor. Test to find the nearest parallel combination of standard resistor values to match \( R = 40.5\,\Omega \). How close is it? 

**PRACTICE PROBLEMS**

**tic, toc**

The commands **tic, toc** will time how long it takes Matlab to execute the commands that separate them. These commands must be either within a function, or cut and pasted into the command window; otherwise, most of the delay between the “tic” and the “toc” will be from your typing speed, not Matlab’s calculation speed limitations.

```matlab
function calculationspeed
% find how long it takes to create two random 50x50
% matrices and multiply them together
 tic
  a = rand(50,50); % create two random 50x50 matrices
  b = rand(50,50);
  c = a*b; % multiply them together
  toc
```

Matlab then returns the following on my 2017-era computer:

```
Elapsed time is 0.278700 seconds.
```

The `toc` command will also return the elapsed time in seconds to a variable.
PRACTICE PROBLEMS

10. How quickly can Matlab solve a set of 100 equations with 100 unknowns? This problem was described in Chapter 2, but to avoid typing 10,000 test coefficients, create a random problem. First create a 100x100 matrix of random numbers and save in variable A using the command:

   \[ A = \text{rand}(100,100); \]

Then create a 100 x 1 column vector of random numbers called b. Finally, time how long it takes to solve them using the command:

   \[ V = A\backslash b; \]

FUNCTIONS CALLING FUNCTIONS

Functions frequently call other functions inside them, as shown below:

   \[
   \text{function } z = \text{polar2complex}(\text{mag}, \text{angle})
   \]

   \% polar2degrees returns a complex number given
   \% the magnitude and angle in degrees
   \% z = \text{mag} \cdot \text{cosd}(\text{angle}) + j \cdot \text{mag} \cdot \text{sind}(\text{angle});

The \text{sind()} and \text{cosd()} are built-in functions called inside \text{polar2complex}. User-created functions can be called the same way. Consider the following function that returns the equivalent resistance of two resistors in parallel:

   \[
   \text{function } p = \text{parallel}(r1, r2)
   \]

   \% parallel returns the parallel resistance
   \% of two given resistors
   \% p = (r1 \cdot r2) / (r1 + r2);

The parallel function can be called by other user-defined functions. For example the Monte-Carlo routines described on page 164 could call \text{parallel} repeatedly in its simulations, allowing the Monte-Carlo code to become more readable.
**break**

It is sometimes necessary to break out of the middle of a loop before it is completed. For example, this might happen if the loop is searching for some criteria to be met, such as trying to minimize a calculated error below some threshold. Once this is accomplished, the loop has served its purpose; in such a case, you can exit the loop immediately using the break command.

Another example involves searching to see if a given number is prime. One method, given a number \( n \), is to loop over all possible integers between 2 and \( n \) to see if any are factors of the given number. The moment any factors are found, the loop exits and reports that the number is not prime. If the loop completes with no factors found, the function reports that the number must be prime. But rather than searching for possible factors from 2 through \( n \), one only needs to search up to the square root of \( n \), since any factor greater than this must be multiplied by a factor less than this.

Define the following helper function and save as isInteger.m:

```matlab
function b = isInteger(x)
    b = x == floor(x) % b is 1 if x is an integer, % otherwise 0
end
```

Last, define the main function to locate primes, and save as isPrimes.m:

```matlab
function b = isPrime(n)
    b = 1; % assume it is prime, then check
    for f=2:floor(sqrt(n)) % f cycles through % all possible factors
        % code to check if f is a factor of n
        if rem(n,f) == 0
            b = 0; % n is not prime
            break
        end
    end
end
```
if isInteger(n/f) % is f a factor of n?
    b = 0; % change b to 0
    break; % break out of the loop
end
end
This program would run correctly without the break statement, but it would be slower; if a factor were found, it would continue to run the loop until the end, rather than returning the result immediately.

PRACTICE PROBLEMS

11. Change the above code in isPrime to check to make sure that the program has not been running for over 1 second. Do this by starting a timing operation at the beginning of the code using \texttt{tic}, and check the elapsed time inside the loop with \texttt{toc}. Use the \texttt{break} statement to exit if the code has been executing for over 1 second, and if so exit and return a -1. This is a useful technique to exit automatically when a simulation does not converge on an answer.

MULTIPLE FUNCTIONS IN ONE M-FILE

In the previous example, a separate helper function called \texttt{isInteger} was saved as a separate file, even although its purpose was only to be used within the \texttt{isPrime} function. It is possible to pack multiple helper functions inside the same m-file that houses the main function, as long as the helper functions are defined after the main function. These helper functions can only be called by the main function.
For example, re-writing the last example using this method yields a single file to be saved as isPrime.m:

```matlab
function b = isPrime(n)
b = 1; % assume it is prime, then check
for f=2:floor(sqrt(n)) % f cycles through % all possible factors
    if isInteger(n/f) % is f a factor of n?
        b = 0; % change b to 0 to show n is not prime
        break; % break out of the loop
    end
end
% helper functions
function b = isInteger(x)
b = (x == floor(x)); % b is 1 if x is an integer, % otherwise 0
```

Defined this way, the helper function `isInteger` cannot be seen by the Matlab command line or any function other than `isPrime`.

**while-end loop**

The `for-end` command loops for a number of times that is known in advance, unless a `break` statement ends it early. The `while-end` loop will loop until a logical condition is true.

```matlab
while(logical test statement)
    program code to be run
end
```

It is less common than the `for-end` loop command, and like the `for-end` loop command, can be exited early using the `break` statement. It can be used, for instance, in simulations that iterate until an error is acceptably small.

```matlab
function sum=WhileExample(x)
    % WhileExample will loop until new additions are
    % less than x
```
term = 1;
sum = 0;
while (term >= x) % run until the terms are less than x
    term = term/2;
    sum = sum+term;
end
This function snippet adds the geometric series \( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots \) until the terms become smaller than some given number x.

**PRACTICE PROBLEMS**

12. Change the above code so that it sums the sequence \( \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \ldots \) Report the result for summing until the last term added is smaller than 0.00001.

**GROWING VECTORS VS. PRE-ALLOCATING THEM**

Often a function will return a vector whose elements are calculated inside a loop. In this case the vector may be either initialized to the full size with a `zeros()` command and then filled inside the loop, or else grown to size inside the loop.

**DIGGING DEEPER**

The reason why vectors should be initialized with some value like zeros, rather than grown as needed, is because of the way Matlab grows vectors. If a vector needs be grown to accommodate a new value, for instance to go from length 100 to length 101, Matlab must first internally create an entirely new 101 length vector, then copy the original 100 values over to it, then save the additional single value, and then destroy the original length 100 vector. Compare this with the faster alternative of preallocating memory for the final array once, and then replacing the initialized values with the calculated values.
with its length increasing with every loop. The former method much more efficient, and is accomplished by reserving the required memory for the vector with a command like this:

\[
\text{result} = \text{zeros}(1,1000);
\]

The latter less-efficient method is to grow the size of vector in the middle of the loop, like this:

\[
\text{result} = [\text{newvalue} \ \text{result}];
\]

If the resulting vector size is known before the loop begins executing, as if often the case, use the former method. Only if the resulting vector length is not known in the advance should the latter method be used.

**Good practice: preallocate memory**

```matlab
function v = GoodProgram(n)
% return a result of length n and preallocate
% the memory needed
v = zeros(1,n);
for i=1:n
    v(n) = rand + sin(2*i);
end
```

**Bad practice: grow memory**

```matlab
function v = BadProgram(n)
% return a result of length n but grow the memory
% as needed
v=[];  % define v as being empty
for i=1:n
    newval = rand + sin(2*i);  % newval is calculated
    v = [newval v];  % insert newval at the start
                        % of the old vector
end
```
TECH TIP: BINARY MATH WITH MATLAB

There are two types of mathematics electrical engineers do with 0 and 1: logical and binary.

Logical, or Boolean Math
These use numbers to represent true (1) and false (0). The logical operations of &&, ||, and ~ model the effects of their counterpart logic gates AND, OR, and NOT gates.

As an example, find the output of the following digital circuit with Matlab:

The output can be found by substituting the Matlab logical operations, as follows:

The Matlab code to calculate the answer given the inputs therefore is ~(1||0)&&1, which evaluates to 0.
**Binary math:** `dec2bin()` and `bin2dec()`

Binary mathematics involves calculations with numbers in base 2, but Matlab shows all numbers in base 10. To convert to base 2 use `dec2bin()`

For example, to convert 77 to base 2:

```
    dec2bin(77)  returns  1001101
```

To convert back into base 10, use `bin2dec` and put string markers around the binary value

```
    bin2dec('1001101')  returns  77
```

As an example, perform the following operation in Matlab:

```
    10011_2 x 1001_2 + 1010_2:
    bin2dec('10011') * bin2dec('1001') +
    bin2dec('1010')
```

This returns 181 (in base 10). To see the result in binary, use:

```
    dec2bin(181)
```

This returns:

```
    10110101 (in base 2)
```

**PRACTICE PROBLEMS**

13. Use Matlab to calculate the following logic block ©®

```
  0
  1
  1
```

14. Use Matlab to evaluate the following binary math problem, and report the answer in binary: \((1001_2 + 1110_2) \times 10011_2\) ©®
PRO TIP: FUNDAMENTALS OF ENGINEERING

An earlier Pro Tip discussed the advantages to taking the optional step of earning the Professional Engineering license. The first step in becoming a PE is to pass the Fundamentals of Engineering (FE) exam. This test is designed to be taken during the senior year, or by recent graduates of an ABET-accredited, four-year engineering program. There are versions of the FE exam for Electrical, Mechanical, Civil, and other engineering majors; the FE exam includes topics common to all of these majors, such as mathematics, probability, ethics, and engineering economics, as well as topics unique to the selected engineering discipline. The EE exam, for instance, also includes circuit analysis, signal processing, electromagnetics, digital systems, and programming. The exam is composed of two three-hour, computer-graded segments, which are taken with a break for lunch. Certain models of scientific non-programmable calculators are authorized, and a book containing reference equations is provided. You can find out more about the exam, how to purchase a copy of the reference equation book, and how to register for it by visiting NCEES.org, the administering testing agency.

Although scores are reported, it is essentially a pass/fail exam since its only purpose is to serve as one of several steps required to become eligible to take the PE exam. In past years roughly 70% of electrical and computer engineering test-takers earned passing scores, and one can retake the exam as many times as required, up to once in every two-month testing window, and up to three times per year.
Very few undergraduate majors have a national licensing examination as engineers do. Although most electrical engineering careers do not require the PE, it is difficult to determine as a student where your particular career path will lead. Why not take advantage that national licensure exists for electrical engineers, and gain maximum career flexibility and a distinguishing résumé line by planning to take it FE as a senior?
COMMAND REVIEW

Looping Functions

for-end   for i=1:10, fprintf('Loop %g', i); end
break     breaks out of a loop
while(condition)-end   loop until the logical condition is met, then exit

Utility Functions

tic, toc   prints elapsed time

tic, t=toc;   saves elapsed time in seconds in variable t
disp()

Binary Functions

dec2bin(73)   converts a base 10 value into base 2

bin2dec('101')   converts a base 2 number (in quotes) into base 10
LAB PROBLEMS

1. Write a Matlab program that takes vector \( t \), and returns a vector \( v \), according to the formula:

\[
v(t) = \begin{cases} 
1 + t, & t \leq 0 \\
2e^{-t} - 1, & t > 0 
\end{cases}
\]

Use a for-end loop in your solution. The function definition line is:

```matlab
function v = problem1(t)
```

Test your code by verifying in the command line that \( \text{problem1}(-5) \) is \(-4\) and \( \text{problem1}(1) \) evaluates to \(-0.2642\).

2. A compander is a device that passes audio signals from -1V to 1V without change, but reduces the gain of higher magnitude signals, in part to prevent very large transient (short length) signals from saturating the system. You will learn more about compressors in Communication courses.

An example of a compander is the rule:

\[
v_{\text{out}}(v_{\text{in}}) = \begin{cases} 
\frac{1}{2}v_{\text{in}} - \frac{1}{2}, & -2 \leq v_{\text{in}} \leq -1 \\
v_{\text{in}}, & -1 < v_{\text{in}} < 1 \\
\frac{1}{2}v_{\text{in}} + \frac{1}{2}, & 1 \leq v_{\text{in}} \leq 2
\end{cases}
\]

Note that the horizontal axis (the “x” axis) is \( v_{\text{in}} \) and the vertical axis (the “y” axis) is \( v_{\text{out}} \). Write a Matlab program that takes a vector of values for \( v_{\text{in}} \) and returns the associated \( v_{\text{out}} \) vector. It should have the following function definition line:

```matlab
function vout = problem2(vin)
```

Test the code by verifying in the command line that \( \text{problem2}(-2) \) is \(-1.5\), \( \text{problem2}(0) \) is \(0\), and \( \text{problem2}(2) \) evaluates to \(1.5\).

3. In the “Nested Loops” section on page 167, there is a code segment called “FindResistors” that outputs all combinations of 4 resistors taken 2 at a time. However, it does not recognize that \( R1 = 10, R2 = 22 \) is the same combin-
tion as $R_1 = 22$, $R_2 = 10$. Modify the code to avoid listing these duplicates. Hint: change the start index of $i_2$ from 1 to something else. This requires some thought; it may help to find the pattern by crossing out the repeats in the output table. It should have the following function definition line:

```matlab
function problem3()
```

4. The “Nested Loops to Search for Best Solutions” section on page 171 has code to find the two standard-value resistors whose value in series is the nearest to a given desired value. The associated Practice Problem asks you to modify that to return the two standard-value resistors whose value in parallel is the nearest to a given desired value. Modify the code so that it returns the best single resistor, or set of two resistors in series or parallel, that is closest to the given value. The function definition is:

```matlab
[R1best, R2best, result] = problem4(rDesired)
```

The result is the equivalent resistance computed with $R_{1\text{best}}$ and $R_{2\text{best}}$, which shows both how close the match to $r_{\text{desired}}$ is and whether $R_{1\text{best}}$ and $R_{2\text{best}}$ should be connected in series or parallel. If the closest match is a single resistor, then $R_{2\text{best}}$ should be 0.

Test the code by verifying in the command line that one can create an 8.5 ohm resistor by placing a 1 ohm and a 7.5 ohm resistor in series, equivalently

```matlab
[r1, r2, exact] = problem4(8.5)
```
evaluates to

```
r1 = 1.0, r2 = 7.5, exact = 8.5
```

Similarly verify that the best way to create a 2.555 ohm resistor is by placing a 4.7 ohm and 5.6 ohm resistor in parallel, equivalently

```matlab
[r1, r2] = problem4(2.555)
```
evaluates to

```
r1 = 4.7, r2 = 5.6, exact = 2.5553
```

5. The following sum will calculate $\pi$ in the limit as $N$ grows large.

$$f(N) = \sum_{k=0}^{N} \frac{\sqrt{12}}{(-3)^k (2k + 1)}$$
Calculate it by writing a function using the following function definition:

```matlab
function result = problem5(N)
```
To test it, setting N=7 will return π to 4 decimal places of accuracy.

6. Use Matlab to write a function that solves the math problem:

```matlab
result = (x + y) * 7
```
The function definition is:

```matlab
function result = problem6(x,y)
```
The result should be in base 2. The arguments will be in base 2 and should be provided with quotes. To check your work:

```matlab
problem6('101','110')
```
should return:

```
1001101
```

7. Fourier series analysis is an important technique from the signal processing subdiscipline of electrical engineering. It involves the fact that any periodic waveform can be broken up into a sum of sinusoids. It sounds impossible that a sharp-edged function like a square wave could be composed of a sum of smooth sinusoids (indeed, Fourier's original 1807 paper on the subject was denied publication for this reason) but in the limit of summing an infinite number of sinusoids, it is true. As an example, create a function with the following function definition

```matlab
function f = problem7()
```
that calculates the following sum with \( t \) being a 1000 element time vector ranging from \( 0 \leq t \leq 20 \) (\( f \) and \( t \) are bolded because they are vectors):

\[
f = \sum_{k=1,3,5,...}^{49} \frac{\sin(kt)}{k}
\]

In words, this means to first create a time vector \( t \), and then initialize the result vector \( f \) to be filled with a vector of zeros that is the same size as the \( t \) vector. Next, create a for loop using loop variable \( k \) that starts at 1 and increments in steps of 2 so it becomes every odd number up to 49. For every iteration of the loop, add the vector \( \frac{\sin(kt)}{k} \) to \( f \). This is a vector because \( t \) is a vector. The function returns \( f \), but check your work by embedding a
plot(t, f) at the end of your code. It should look similar to a squarewave. As the upper limit of the sum, 49, is increased the function will appear progressively more similar to an ideal square wave.

8. Challenging!: You want to create a 72Ω resistor, but that is not a standard value. One way to build it is with a 33Ω and 39Ω in series. Another way is with a 75Ω and 1800Ω in parallel. Numerically, both these options work out to the exact same 72Ω resistor. If you do a Monte Carlo analysis of the two options with 10,000 simulations, do the two options have the same spread of tolerances if they all made with 5% resistors? For this problem, provide only the two histograms of data, labeled to show which is the series and which is the parallel solution. Also, state which is the better method or if both are equally accurate.