Chapter 19

Morse Code

Morse code demonstrates binary trees and cell arrays.

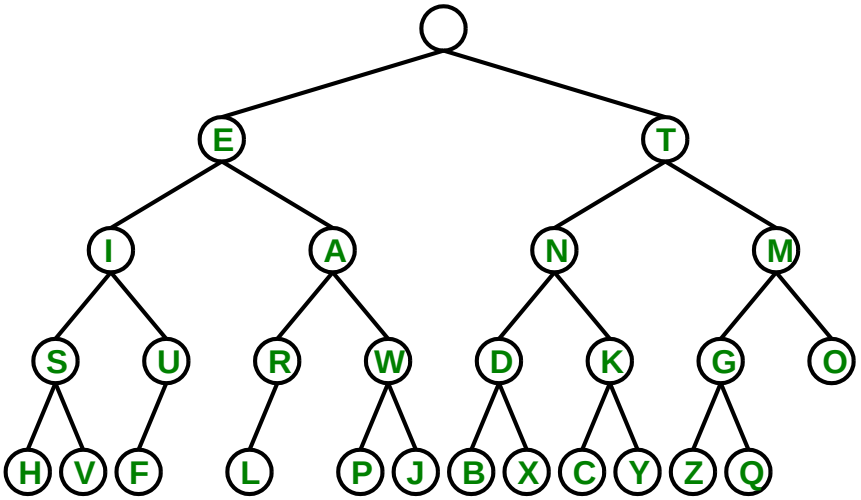

Figure 19.1. *The binary tree defining Morse code. A branch to the left signifies a dot in the code and a branch to the right is a dash. In addition to the root node, there are 26 nodes containing the capital letters of the English alphabet.*

This chapter brings together three disparate topics: Morse code, binary trees, and cell arrays. Morse code is no longer important commercially, but it still has

October 2, 2011

some avid fans among hobbyists. Binary trees are a fundamental data structure used throughout modern computing. And cell arrays, which are unique to MATLAB, are arrays whose elements are other arrays.

You can get a head start on our investigation by running the `exm` program

```
morse_gui
```

Experiment with the four buttons in various combinations, and the text and code box. This chapter will explain their operation.

Morse code was invented over 150 years ago, not by Samuel F. B. Morse, but by his colleague, Alfred Vail. It has been in widespread use ever since. The code consists of short *dots*, `'.'`, and longer *dashes*, `'-'`, separated by short and long spaces. You are certainly familiar with the international distress signal, `'... --- ...'`, the code for "SOS", abbreviating "Save Our Ships" or perhaps "Save Our Souls". But did you notice that some modern cell phones signal `'... -- ...'`, the code for "SMS", indicating activity of the "Short Message Service".

Until 2003, a license to operate an amateur radio required minimal proficiency in Morse code. (Full disclosure: When I was in junior high school, I learned Morse code to get my ham license, and I've never forgotten it.)

According to Wikipedia, in 2004, the International Telecommunication Union formally added a code for the ubiquitous email character, `@`, to the international Morse code standard. This was the first addition since World War I.

The Morse tree

We could provide a table showing that `'.-'` is the code for A, `'-...'` the code for B, and so on. But we're not going to do that, and our MATLAB program does not start with such a table. Instead, we have figure 19.1. This is a *binary tree*, and for our purposes, it is the *definition* of Morse code. In contrast to nature, computer scientists put the root of a tree on top. Starting at this root, or any other node, and moving left along a link signifies a *dot*, while moving right is a *dash*. For example, starting at the root and moving one step to the left followed by one to the right gets us to A. So this fact, rather than a table, tells us `'.-'` is the code for A.

The length of a Morse code sequence for a particular character is determined by the frequency of that character in typical English text. The most frequent character is "E". Consequently, its Morse sequence is a single dot and it is linked directly to the root of our tree. The least frequent characters, such as "Z" and "X", have the longest Morse sequences and are far from the root. (We will consider the four missing nodes in the tree later.)

Binary trees are best implemented in MATLAB by *cell arrays*, which are arrays whose elements are themselves other MATLAB objects, including other arrays. Cell arrays have two kinds of indexing operations. Curly braces, `{` and `}`, are used for construction and for accessing individual cells. Conventional smooth parentheses, `(` and `)`, are used for accessing subarrays. For example,

```
C = {'A','rolling','stone','gathers','momentum','.'}
```

produces a cell array that contains six strings of different lengths. This example is displayed as

```
C =
   'A' 'rolling' 'stone' 'gathers' 'momentum' '.'
```

The third *element*, denoted with curly braces by `C{3}`, is the string `'stone'`. The third *subarray*, denoted with parentheses by `C(3)`, is another cell array containing a single element, the string `'stone'`. Now go back and read those last two sentences a few more times. The subtle distinction between them is both the key to the power of cell arrays and the source of pervasive confusion.

Think of a string of mailboxes along `C` street. Assume they are numbered consecutively. Then `C(3)` is the third mailbox and `C{3}` is the mail in that box. By itself, `C` is the entire array of mailboxes. The expression `C(1:3)` is the subarray containing the first three mailboxes. And here is an unusual construction, with curly braces `C{1:3}` is a *comma separated list*,

```
C{1}, C{2}, C{3}
```

By itself, on the command line, this will do three assignment statements, assigning the contents of each of the first three mailboxes, one at a time, to `ans`. With more curly braces, `{C{1:3}}` is the same as `C(1:3)`.

In the computer hardware itself, there is a distinction between a memory location with a particular address and the contents of that location. This same distinction is preserved by indexing a cell array with parentheses and with braces.

Did you see the "Men in Black" movies? In one of them the MIB headquarters has a bank of storage lockers. It turns out that each locker contains an entire civilization, presumably with its own lockers. At the end of the movie it is revealed that the Earth itself is a storage locker in a larger civilization. It is possible that we are all living in one element of a huge cell array.

The binary tree defining Morse code is a cell array whose contents are characters and other cell arrays. Each cell represents a node in the tree. A cell, `N`, has three elements, The first element, `N{1}`, is a string with a single capital letter, `X`, designating the node. The second element, `N{2}`, is another cell array, the *dot* branch. The third element, `N{3}`, is the *dash* branch. There are a couple of exceptional cases. The root node does not have an associated letter, so its first element is an empty string. The `U` and `R` nodes, and the leaf nodes, have one or two empty cell arrays for branches.

In principle, we could create the entire Morse binary tree with a single gigantic, but unrealistic, assignment statement.

```
M = {'' ...
    {'E' ...
       {'I' {'S' {'H' {} {}} {'V' {} {}}} ...
            {'U' {'F' {} {}} {}}} ...
       {'A' {'R' {'L' {} {}} {}} ...
            {'W' {'P' {} {}} {'J' {} {}}}}} ...
    {'T' ...
```

```
{'N' {'D' {'B' {} {}} {'X' {} {}}} ...
     {'K' {'C' {} {}} {'Y' {} {}}}} ...
{'M' {'G' {'Z' {} {}} {'Q' {} {}}} ...
     {'O' {} {}}}}
```

You can see the cell arrays within cell arrays and the many empty cell arrays at the leaves. This statement actually works, but it is unreasonable because it is error prone and nearly impossible to extend. Instead, our function `morse_tree` begins with a header

```
function M = morse_tree
```

This is followed by 27 assignment statements, twelve at level four.

```
h = {'H' {} {}};
v = {'V' {} {}};
f = {'F' {} {}};
l = {'L' {} {}};
p = {'P' {} {}};
j = {'J' {} {}};
b = {'B' {} {}};
x = {'X' {} {}};
c = {'C' {} {}};
y = {'Y' {} {}};
z = {'Z' {} {}};
q = {'Q' {} {}};
```

Eight at level three.

```
s = {'S' h v};
u = {'U' f {}};
r = {'R' l {}};
w = {'W' p j};
d = {'D' b x};
k = {'K' c y};
g = {'G' z q};
o = {'O' {} {}};
```

Four at level two.

```
i = {'I' s u};
a = {'A' r w};
n = {'N' d k};
m = {'M' g o};
```

Two at level one.

```
e = {'E' i a};
t = {'T' n m};
```

And finally one assignment statement at level zero to create the root node.

```
M = {'' e t};
```

This function is at the heart of all our Morse code software.

You can travel down this tree by first entering three commands.

```
M = morse_tree
M = M{2}
M = M{3}
```

You then can use the up-arrow on your keyboard to repeatedly select and reexecute these commands. Returning to the first command gives you a fresh tree. Executing the second command is a *dot* operation, moving down the tree to the left. Executing the third command is a *dash* operation, moving down the tree to the right. For example, the five commands

```
M = morse_tree
M = M{3}
M = M{2}
M = M{2}
M = M{3}
```

correspond to the Morse sequence `'-..-'`. This brings you to the node

```
'X'    {}    {}
```

You have reached the `X` leaf of the tree.

Searching the tree

Returning to the "Men in Black" analogy, the Morse binary tree is a single locker. When you open that locker, you see an empty string (because the root does not have a name) and two more lockers. When you open the locker on the left, you see an `'E'` and two more lockers. Now you have three choices. You can open either of the two lockers in the `E` locker, or you can go back to the root locker and open the locker on the right to visit the `T` locker.

Repeatedly choosing different lockers, or different branches in the tree, corresponds to traversing the tree in different orders. Among these many possible orders, two have standard names, "depth first search" and "breadth first search". They are shown in figures 19.2 and 19.3 and you can see and hear animated versions with the `morse_gui` program.

Depth first search visits each branch as soon as it sees it. When it has visited both branches at a node, it backs up to the first node that has an available branch. Figure 19.2 shows the progress of depth first order up to node `W`.

```
E I S H V U F A R L W
```

Nothing to the right of `W` has yet been visited.

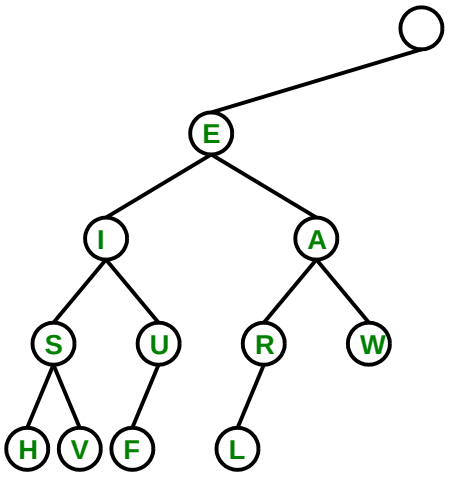

Figure 19.2. *A depth first search in progress. The nodes are visited from left to right.*

Breadth first search takes one step along each branch at a node before it continues. The result is a top to bottom search, much like reading English language text. Figure 19.3 shows the breadth first order up to node `W`.

```
E T I A N M S U R W
```

Nothing below `W` has yet been visited.

Depth first search uses a data structure known as a *stack*. Here is a code segment with a stack that simply displays the nodes of the tree in depth first order.

```
S = {morse_tree};
while ~isempty(S)
   N = S{1};
   S = S(2:end);
   if ~isempty(N)
      fprintf(' %s',N{1})
      S = {N{2} N{3} S{:}};
   end
end
fprintf('\n')
```

The stack `S` is a cell array. Initially, it has one cell containing the tree. The `while`

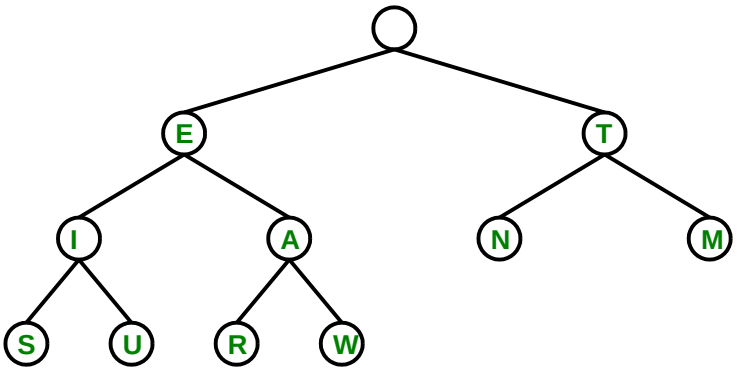

Figure 19.3. *A breadth first search in progress. The nodes are visited from top to bottom.*

loop continues as long as the stack is not empty. Within the body of the loop a node is removed from the top of the stack and the stack shortened by one element. If the node is not empty, the single character representing the node is displayed and two new nodes, the *dot* and *dash* branches, are inserted into the top of the stack. The traversal visits recently discovered nodes before it returns to older nodes.

Breadth first search uses a data structure known as a *queue*. Here is another code segment, this time with a queue, that displays the nodes of the tree in breadth first order.

```
Q = {morse_tree};
while ~isempty(Q)
   N = Q{1};
   Q = Q(2:end);
   if ~isempty(N)
      fprintf(' %s',N{1})
      Q = {Q{:} N{2} N{3}};
   end
end
fprintf('\n')
```

This code is similar to the stack code. The distinguishing feature is that new nodes

are inserted at the end, rather that the beginning of the queue.

A queue is employing a "First In, First Out", or FIFO, strategy, while a stack is employing a "Last In, First Out", or LIFO, strategy. A queue is like a line at a grocery store or ticket office. Customers at the beginning of the line are served first and new arrivals wait at the end of the line. With a stack, new arrivals crowd in at the start of the line.

Our stack and queue codes are not recursive. They simply loop until there is no more work to be done. Here is different approach that employs recursion to do a depth first search. Actually, this does involve a hidden stack because computer systems such as MATLAB use stacks to manage recursion.

```
function traverse(M)
   if nargin == 0
      M = morse_tree;    % Initial entry.
   end
   if ~isempty(M)
      disp(M{1})
      traverse(M{2})     % Recursive calls.
      traverse(M{3})
   end
end % traverse
```

Decode and encode

Decoding is the process of translating dots and dashes into text. Encoding is the reverse. With our binary tree, decoding is easier than encoding because the dots and dashes directly determine which links to follow. Here is a function that decodes one character's worth of dots and dashes. The function returns an asterisk if the input does not correspond to one of the 26 letters in the tree.

```
function ch = decode(dd)
   M = morse_tree;
   for k = 1:length(dd)
      if dd(k) == '.'
         M = M{2};
      elseif dd(k) == '-'
         M = M{3};
      end
      if isempty(M)
         ch = '*';
         return
      end
   end
   ch = M{1};
end % decode
```

Encoding is a little more work because we have to search the tree until we find the desired letter. Here is a function that employs depth first search to encode one character. A stack of dots and dashes is built up during the search. Again, an asterisk is returned if the input character is not in the tree.

```
function dd = encode(ch)
   S = {morse_tree};
   D = {''};
   while ~isempty(S)
      N = S{1};
      dd = D{1};
      S = S(2:end);
      D = D(2:end);
      if ~isempty(N)
         if N{1} == ch;
            return
         else
            S = {N{2} N{3} S{:}};
            D = {[dd '.'] [dd '-'] D{:}};
         end
      end
   end
   dd = '*';
end % encode
```

These two functions are the core of the decoding and encoding aspect of `morse_gui`.

Extension

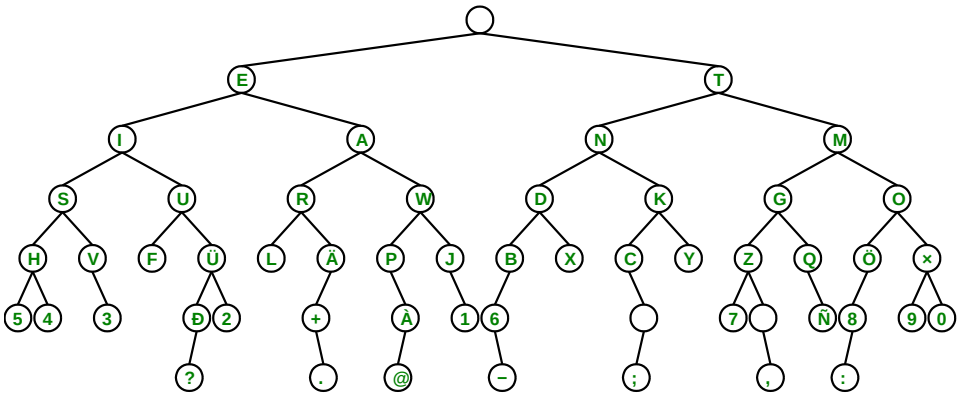

Figure 19.4. *The Morse tree extended two levels to accommodate digits, punctuation, and non-English characters.*

A binary tree with four levels has $2^5 - 1 = 31$ nodes, counting the root. Our `morse_tree` has a root and only 26 other nodes. So, there are four empty spots. You can see them on the dash branches of U and R and on both branches of O. So far, our definition of Morse code does not provide for the four sequences

```
..--   .-.-   ---.   ----
```

How should these be decoded? We also want to add codes for digits and punctuation marks. And, it would be nice to provide for at least some of the non-English characters represented with umlauts and other diacritical marks.

Morse code was invented 100 years before modern computers. Today, alphabets, keyboards, character sets and fonts vary from country to country and even from computer to computer. Our function

```
morse_table_extended
```

extends the tree two more levels and adds 10 digits, 8 punctuation characters, and 7 non-English characters to the 26 characters in the original `morse_table`. Figure 19.4 shows the result. The `extend` button in `morse_gui` accesses this extension. Some of the issues involved in representing these additional characters are pursued in the exercises.

Morse code table

We promised that we would not use a table to define Morse code. Everything has been based on the binary tree. When an actual table is desired we can generate one from the tree. Our function is named `morse_code` and it employs the recursive algorithm `traverse` from the previous section. The recursion carries along the emerging table `C` and a growing string `dd` of dots and dashes.

Indexing into the table is based upon the ASCII code for characters. The ASCII standard is an 7-bit code that is the basis for computer representation of text. Seven bits provide for $2^7 = 128$ characters. The first 32 of these are nonprinting characters that were originally used for control of teletypes and are now largely obsolete. The remaining 96 are the 52 upper and lower case letters of the English alphabet, 10 digits, and 32 punctuation marks. In MATLAB the function `char` converts a numeric value to a character. For example

```
char(65)
ans =
   'A'
```

The function `double` converts a character to a floating point number and the function `uint8` converts a character to an unsigned 8-bit integer. Either of these can be used as an index. For example

```
double('A')
ans =
   65
```

A computer byte is 8 bits and so the ASCII standard is readily extended by another 128 characters. The actual graphic printed for some of these characters may vary from country to country and from font to font.

Our function `morse_code` produces a table of both ASCII and Morse code from either `morse_tree` or `morse_tree_extended`. The recursive `traverse` algorithm is used for the depth first search. ASCII codes are used as indices into a 256-element cell array of dots and dashes. The search inserts only 26 or 51 elements into the array, so a final step creates a printable table.

```
function C = morse_code(C,M,dd)
   % MORSE_CODE
   % C = morse_code
   % C = morse_code(morse_tree)
   % C = morse_code(morse_tree_extended)

      if nargin < 3           % Choose binary tree
         if nargin == 0
            M = morse_tree;
         else
            M = C;
         end
         C = cell(256,1);     % The temporary code table
         dd = '';             % dots and dashes
      end

      if ~isempty(M)                        % Depth first search
         if ~isempty(M{1})
            C{double(M{1})} = dd;           % Use ASCII value as an index
         end
         C = morse_code(C,M{2},[dd '.']);   % Recursive call
         C = morse_code(C,M{3},[dd '-']);   % Recursive call
      end

      if nargin < 3                    % Final processing, convert to char.
         c = char(C{:});
         k = find(c(:,1) ~= ' ');      % Find the nonblank entries.
         b = blanks(length(k))';
         C = [char(k) b b int2str(k) b b char(C{k})];
      end
```

The output from

```
   morse_code
```

is the following 26 lines.

```
      A   65  .-
      B   66  -...
```

```
C   67  -.-.
D   68  -..
E   69  .
F   70  ..-.
G   71  --.
H   72  ....
I   73  ..
J   74  .---
K   75  -.-
L   76  .-..
M   77  --
N   78  -.
O   79  ---
P   80  .--.
Q   81  --.-
R   82  .-.
S   83  ...
T   84  -
U   85  ..-
V   86  ...-
W   87  .--
X   88  -..-
Y   89  -.--
Z   90  --..
```

You should run

```
morse_code(morse_tree_extended)
```

to see what output is generated on your computer.

References

[1] Wikipedia article on Morse code. See the complete Morse code tree at the end of the article.

```
http://en.wikipedia.org/wiki/Morse_code
```

Recap

```
%% Morse Code Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Morse Code Chapter of "Experiments in MATLAB".
% You can access it with
%
%    morse_recap
%    edit morse_recap
```

```
%    publish morse_recap
%
% Related EXM programs
%
%    morse_gui
%    morse_tree
%    morse_tree_extended
%    morse_code

%% Cell Arrays

   C = {'A','rolling','stone','gathers','momemtum','.'}
   C{3}
   C(3)
   C(1:3)
   C{1:3}
   {C{1:3}}

%% Create a Morse Tree
% An absurd statement.  For a better way, see morse_tree.

   M = {'' ...
       {'E' ...
          {'I' {'S' {'H' {} {}} {'V' {} {}}} ...
               {'U' {'F' {} {}} {}}} ...
          {'A' {'R' {'L' {} {}} {}} ...
               {'W' {'P' {} {}} {'J' {} {}}}}} ...
       {'T' ...
          {'N' {'D' {'B' {} {}} {'X' {} {}}} ...
               {'K' {'C' {} {}} {'Y' {} {}}}} ...
          {'M' {'G' {'Z' {} {}} {'Q' {} {}}} ...
               {'O' {} {}}}}}

%% Follow '-..-'
   M = morse_tree
   M = M{3}
   M = M{2}
   M = M{2}
   M = M{3}

%% Depth first, with a stack.
   S = {morse_tree};
   while ~isempty(S)
      N = S{1};
      S = S(2:end);
      if ~isempty(N)
```

```
         fprintf(' %s',N{1})
         S = {N{2} N{3} S{:}};
      end
   end
   fprintf('\n')

%% Breadth first, with a queue.
   Q = {morse_tree};
   while ~isempty(Q)
      N = Q{1};
      Q = Q(2:end);
      if ~isempty(N)
         fprintf(' %s',N{1})
         Q = {Q{:} N{2} N{3}};
      end
   end
   fprintf('\n')

%% Recursive traversal.
%   function traverse(M)
%      if nargin == 0
%         M = morse_tree;    % Initial entry.
%      end
%      if ~isempty(M)
%         disp(M{1})
%         traverse(M{2})     % Recursive calls.
%         traverse(M{3})
%      end
%   end % traverse

%% ASCII character set
   k = reshape([32:127 160:255],32,[])';
   C = char(k)
   txt = text(.25,.50,C,'interp','none');
   set(txt,'fontname','Lucida Sans Typewriter')
```

Exercises

19.1 *Greetings.* What does this say?

```
.... . .-.. .-.. ---   .-- --- .-. .-.. -..
```

19.2 *Email.* Use the `extend` button and translate box in `morse_gui` to translate your email address into Morse code.

19.3 *dash.* Why didn't I use an underscore, `'_'`, instead of a minus sign, `'-'`, to represent a dash?

19.4 *Note.* What musical note is used by the sound feature in `morse_gui`?

19.5 *Reverse order.* Find this statement in function `depth` in `morse_gui`.

```
S = {N{2} N{3} S{:}};
```

What happens if you interchange `N{2}` and `N{3}` like this?

```
S = {N{3} N{2} S{:}};
```

19.6 *Extend.* Using the tree at the end of the Wikipedia article, add more characters to `morse_code_extended`. If you live in a *locale* that has non-English characters in the alphabet, be sure to include them.

19.7 *Four dashes.* Why does the character with Morse code `'----'` cause a unique difficulty in `morse_code_extended`?

19.8 *YouTube.* Check out "Morse code" on YouTube. Be sure to listen to the "Morse code song". Who wins the Morse code versus texting contest on Jay Leno's Tonite show?

19.9 *Trinary.* What does this function do? Why is it named `trinary`. What determines how long it runs? What causes it to terminate? Why is the `if` statement necessary? Modify the program to make it use a depth first search. Modify the program to make it work with `morse_tree_extended`.

```
function trinary
   T = [0 0 0 0];
   Q = {morse_tree};
   while any(T(1,:) <  2)
      p = T(1,:);
      y = polyval(p,3);
      if ~isempty(Q{1})
         fprintf('%s  %d%d%d%d  %2d\n',Q{1}{1},p,y)
         Q = {Q{2:end} Q{1}{2} Q{1}{3}};
      else
         Q = {Q{2:end} {} {}};
      end
      T = [T(2:end,:); [T(1,2:end) 1]; [T(1,2:end) 2]];
   end
end % trinary
```

19.10 *Cell arrays.* Let

```
C = {'A' 'rolling' 'stone' 'gathers' 'momentum' '.'}
```

Explain why each of the following does what it does.

```
C'
char(C)
size(C)
size(char(C))
double(char(C))
upper(C)
C(end)
C{end}
C{3}(3)
fliplr(C)
[C{:}]
C(5:7) = {'no' 'moss' '.'}
```

19.11 *Fonts.* Experiment with various fonts. See the Wikipedia article on the ASCII character set. You can generate a printable character table in MATLAB with

```
k = reshape([32:127 160:255],32,[])';
C = char(k)
```

The first half of this table is standard, but the second half depends upon the fonts you are using. You can change fonts in the command window by accessing the "File" menu, then selecting "Preferences" and "Fonts". Highlight the desktop code font and use the up and down arrow keys. In my opinion the best font for the MATLAB command window is

Lucida Sans Typewriter

You can also display the character table in the figure window with

```
txt = text(.25,.50,C,'interp','none');
```

To change display fonts in the figure window, try commands like these work on your computer.

```
set(txt,'fontname','Lucida Sans Typewriter')
set(txt,'fontname','Courier New')
set(txt,'fontname','Comic Sans MS')
set(txt,'fontname','Wingdings')
set(txt,'fontname','GiGi')
```

Use the font names available under the command window preferences.