Matlab is an interpreted language: to run a Matlab program, the user must be running Matlab. Matlab is itself a program, one that reads the user’s input (for example, the line “[v,e]=eig(m);”) and decides what to do with it. The user’s input does not contain explicit instructions to the CPU (central processing unit); Matlab must convert what the user says into something the CPU understands. The Matlab executable does consist of instructions (coded as zeros and ones) that correspond to instructions the CPU can execute. However, it’s quite unlikely that the people who wrote Matlab sat down and started typing 0 1 1 0 0 0 1 1 0 1 0 0 1 1 1 .... Probably, they wrote it in a compiled language, like C. The compiler is a program that takes human-readable code, rather reminiscent of a Matlab .m file, and converts it into the zeros and ones of a binary executable.

This tutorial will serve only as a first introduction to C, suitable for a light, one-week mini-course. Really to understand C, one should read the book. There is only one worth owning, written by the language’s authors:


In particular, this week’s tutorial will not cover the language features that make C much more adaptable than Matlab, such as pointers (except tangentially), structures, and dynamic memory management.

1. Writing a program that prints something on the screen.

Kernighan and Ritchie observe that “the first program to write is the same for all languages: print the words ‘hello world’.” This will exemplify the skeletal structure of a program in the new language and demonstrate most of the steps required to make any program run.

Begin by entering the program with an editor. Just as Matlab expected a program name to end in .m, the C compiler expects a program name to end in .c, so you should call your program something like hello.c. Unlike in Matlab, the name of the file is not tied to the name of the subroutine inside; in fact, a C file may contain an arbitrary number of subroutines. (The circled notations are not part of the program; don’t enter them.)

```c
/* hello.c
 * DAR 11/03
 * Print "hello world"
 * usage: hello
 */

#include <stdlib.h> /* standard declarations */
#include <stdio.h> /* declarations for standard I/O */

int main(void)
{
    printf("hello, world\n"); /* \n means the newline character */
    return 0; /* required in ANSI C */
}
```

Notes: ①. A comment in C begins with /* and ends at the first instance of */. Line breaks are immaterial (not just with comments). Comments may not be nested.

②. These two lines include certain standard declarations so that the compiler will know what main and printf mean. This particular program would have compiled correctly without them, but it’s a better idea to include them.

③. This line and the next declare main as a function of no arguments (“void”) returning an integer. The analogue in Matlab would be “function i=main.” Other subroutine names are arbitrary, as in Matlab,
but every program must have exactly one subroutine called main, which will be the first subroutine called when the program runs.

4. The open brace begins a block of code, in this case the subroutine. The matching close brace (analogous to end in Matlab) ends the block of code.

5. The string "hello \n" is analogous to the same thing in Matlab but inside single quotation marks instead of double quotation marks. The function printf prints its argument to "standard output," which is usually the screen, but the user can redirect output to a file. The sequence \n stands for the single character “newline.” Experiment to see what happens without it. Every C "statement" must end in a semicolon. Since C does not usually give any meaning to the end of a line, semicolon takes its place in letting the compiler know when one statement ends and the next begins. In Matlab terms, it is as though every line of C automatically ends in "...". The lines that do not end in semicolon are not considered statements.

6. The subroutine main is required by the ANSI standard to return an integer, which is passed back to the shell. The shell understands 0 to mean successful completion and non-zero to mean that the program did not complete successfully, so here we return 0. Continuing the Matlab analogy of 3, this line is analogous to i=0; if i is the Matlab output variable. C does not have output variables; it returns a single value. (Structures and pointers provide mechanisms for returning more complicated things.)

Now to compile and run the program. I'll give the one-step command; a program that consisted of several different .c files would have to be compiled in two steps.

```
physics.cas.usf.edu % cc -o hello hello.c
physics.cas.usf.edu % hello
hello, world
physics.cas.usf.edu %
```

2. Variables

Matlab lets the programmer start using a new variable anytime she wishes. C variables, however, must be declared and defined before use. For this tutorial, the declaration, telling the compiler what type a variable has, and the definition, telling the compiler where to allocate storage for the variable, are the same. Here is a typical declaration:

```c
int i;
```

Optionally, one may (usually) initialize a variable on the declaration line:

```c
int i=42;
```

All declarations in a program block (enclosed in { ... }) must precede the first non-declaration line. The three simple types you will need in C are int, which stores an integer of standard size for the machine (4 bytes on current hardware), char, which stores a single character (1 byte), and double, which stores a floating-point number. All variables in Matlab are of type double, which explains why one doesn’t need to tell Matlab about types. There are other simple types, for example float and short, but they’re less common. Arithmetic performed on an integer variable is automatically truncated, as though one had called floor in Matlab:

```c
int
main(void)
{
    int i=17;
double x, y, z; /* declare three uninitialized doubles */
    x = i/2;
y = i/2.; /* subtle difference */
z = (double)i/2; /* another way to achieve the same thing */
printf("The value of x is %f, but y=%f, and z=%f\n", x, y, z);
    return 0;
}

This example also illustrates how to print a double using printf. There must be one additional argument to printf for each %f in the format string. To print an int, use %d instead of %f. Do not use %d to print a double, even if you know (as here) that it will take an integral value: doing so will pass the wrong number of bytes to printf and may result in a completely baffling error at some apparently random point of execution later in the program. Similarly, you must be careful that the number of percent-sign directives inside the format string matches the number of subsequent arguments. Some alternative formats for printing a double include %e and %g. For more information, consult the Unix manual page on printf:

    physics.cas.usf.edu % man 3 printf

The “3” tells the Unix man command to look in Section 3 (library subroutines) rather than Section 1 (commands that may be issued from the shell).

3. Reading variables from standard input

Here is some code to read one number (the first column, if there is more than one) from standard input while writing its square to standard output:

    /* square.c */
    /* DAR 11/03 */
    /* Write the square of a column of numbers. */
    /* usage: square < input > output */
    /*
    * A line with no number yields a blank line of output.
    */

    #include <stdlib.h>
    #include <stdio.h>

    int
    main(void)
    {
        char buf[1024];  /* room for one line (if exceeded, truncate) */
        double x;        /* store what’s read */

        while(fgets(buf,sizeof(buf),stdin)) { /* loop until no more input */
            if(sscanf(buf, "%lf", &x)==1) /* did we get something? */
                printf("%f\n", x*x);  /* yes: print the square */
            else
                printf("\n");  /* blank line */
        }
        return 0;
    }

This introduces many different ideas. First, char buf[1024] declares buf to be an array of 1024 characters. This is where we’re going to put each successive line of input. The parentheses around the arguments to while and if are mandatory in C (they were optional in Matlab). The fgets(3) function (the 3 says it’s in Section 3 of the manual) takes three arguments. The first tells where to put the characters, the second how many there are, and the third from where to get them: stdin was declared in <stdio.h>. The function returns non-zero (true) while there’s still input to get, and it returns 0 (false) once standard input has reached the end. The sscanf(3) library function mirrors for input what printf(3) does for
output, but there are some differences. The first argument tells it where to look for input: in this case, the buffer, buf. The second argument is a format string, in which %lf tells it to expect a double. Do not use %f here! The third argument tells it where to put the double. The ampersand in front of x results in a pointer to the variable x: it tells where in memory x is located. One can read ampersand as “address of” or “pointer to.” If one left off the ampersand, one would pass to \texttt{scanf(3)} the value of x instead of its address, and the program would die horribly.

On errors in Unix: you may have thought that Matlab error messages were difficult to decipher, but at least they were informative. They told you where the error occurred and roughly what it was. Unix, in contrast, has only three error messages. (Individual programs, of course, may choose to be more helpful.) The first error message is

\begin{center}
\textbf{Segmentation fault}
\end{center}

(on some systems, \textit{“segmentation violation.”}) It means you made a mistake. The second error, \textit{“bus error,”} also means you made a mistake. The ordinary user hopes never to see the third error, but in the exceedingly unlikely event this misfortune should ever befall you, it is \textit{“panic.”} It means that the person who wrote the operating system (Linus Torvalds in our case) made a mistake, and the computer, just before crashing, had the presence of mind (or silicon) to admit it. (I haven’t seen this error in many years on a commercial Unix. On Linux, I’ve seen it mostly when there’s some awful bug in the hardware, and it’s now called \textit{“Oops.”})

In the example above, the code block for the while is enclosed in open \texttt{(\{} and close \texttt{\}) braces, although in this case it didn’t need to be. If there had been another statement after the second printf (at the same indentation as if and else), the braces would have been necessary to tell the compiler what’s inside and what’s outside the while. Similarly, I could have put braces around each of the two blocks of the if statement, but they weren’t required because each block contained only a single statement.

Once you’ve compiled \texttt{square}, you would like to run it on some input. If you just type \texttt{square} (assuming that’s what you called the output of \texttt{cc(1)}), it will sit there waiting for you to type something. Go ahead. When you’re finished supplying all the input lines, you can type “\texttt{CTRL-D}” (that’s control-D).

A more useful way of running the program is to use an editor to make an input file, say \texttt{input}. Then

\begin{verbatim}
physics.cas.usf.edu % square < input
\end{verbatim}

will use your input and type the output to the screen. If you say

\begin{verbatim}
physics.cas.usf.edu % square < input > output
\end{verbatim}

the output will go into the file \texttt{output}, which you may view with an editor or with \texttt{cat(1)} or \texttt{more(1)}. See my 26-page manual.

4. A traveler’s Matlab-C dictionary

This table should get one started.
<table>
<thead>
<tr>
<th>meaning</th>
<th>C</th>
<th>Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>end of statement</td>
<td>;</td>
<td>end of line</td>
</tr>
<tr>
<td>line continuation</td>
<td><em>not needed</em></td>
<td>...</td>
</tr>
<tr>
<td>comment</td>
<td>/<em>...</em>/</td>
<td>%</td>
</tr>
<tr>
<td>begin block</td>
<td>{</td>
<td>if, for, etc.</td>
</tr>
<tr>
<td>end block</td>
<td>}</td>
<td>end</td>
</tr>
<tr>
<td>i holds an integer</td>
<td>int i;</td>
<td>not needed</td>
</tr>
<tr>
<td>x holds a real number</td>
<td>double x;</td>
<td>not needed</td>
</tr>
<tr>
<td>assignment</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>increment i</td>
<td>i=i+1;</td>
<td>i=i+1;</td>
</tr>
<tr>
<td></td>
<td>or i+=1;</td>
<td>or i++;</td>
</tr>
<tr>
<td></td>
<td>or ++i;</td>
<td></td>
</tr>
<tr>
<td>if block</td>
<td>if(test) statement;</td>
<td>if test statement;</td>
</tr>
<tr>
<td></td>
<td>else statement;</td>
<td>else statement;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>for loop</td>
<td>for(i=start; i&lt;=end; i+=step)</td>
<td>for i=start:step:end statement;</td>
</tr>
<tr>
<td></td>
<td>statement;</td>
<td>end</td>
</tr>
<tr>
<td>general for loop</td>
<td>for(once; test; end_each_iteration) statement;</td>
<td></td>
</tr>
<tr>
<td>while loop</td>
<td>while(test) statement;</td>
<td>while test statement;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>do loop</td>
<td>do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>statement;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>while(test);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A statement may be replaced by a program block</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>statement;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>statement;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>equality test</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td>inequality test</td>
<td>!=</td>
<td>“=”</td>
</tr>
</tbody>
</table>
5. Beyond the first week

I’ve left off almost all mention of strings, argument parsing, dynamic memory allocation, vectors, matrices, structures, linked lists, and other derived data types, since to use these correctly requires a thorough understanding of pointers. Bit operations, scope, and static variables and functions are similarly more advanced topics, as are debuggers, optimizers, profilers, makefiles, preprocessors, and source-code-control systems. However, all of these things become easy once one has some experience programming.