MATLAB Guidelines

Style Guidelines

The purpose of Style Guidelines is to help you write code that is easier for a human to understand. This important because:

- well-organized code is easier to debug
- it is easier to remember what a code that you wrote does, and how it does it
- it is easier for someone else to pick up your code and use it.

Below are a series of guideline that I have found useful in writing code. There are several references available online that go into much more detail than below. See for example the "MATLAB Style Guidelines 2.0", written by Richard Johnson (March 2014). They can be found at http://www.mathworks.com/matlabcentral/fileexchange/46056-matlab-style-guidelines-2-0.

File organization

Always write function files (and not script files). There are at least two good reasons for this:

- Variables associated with a function are only defined within the function (that is, they have local scope). Therefore, each time you execute a function, you get a new set of variables. Variables that were set earlier in your MATLAB session will not interfere with your current function.
- Function files can include sub-functions. This is particularly useful since often something you do will require small helper functions. If you write a script file, each of the helper functions will need to be placed into its own file. Over time (for example, over the course of a semester), the number of small files can get quite large and hard to manage.

A function file is one that starts with a `function` statement, such as:

```
function [x] = thomas(n, a, b, c, d);
```

Always surround function outputs with `[]` and inputs with `()`, even if the function consumes no inputs and/or produces no outputs.

Always include a header comment immediately after the `function` statement, such as:

```
% thomas - solve a tridiagonal matrix using the Thomas algorithm
%
%  [a1  c1                  ] [x1]   [d1]
%  [b2  a2  c2              ] [x2]   [d2]
%  [    b3  a3  c3          ] [x3]   [d3]
```

```
%  [                          ] [. ] = [. ]
%  [              ...         ] [. ]   [. ]
%  [                          ] [. ]   [. ]
%  [                  bn  an ] [xn]   [dn]
%
% inputs:
%    n               size of matrix
%    a               array of      diagonal elements
%    b               array of   subdiagonal elements
%    c               array of superdiagonal elements
%    d               array of right-hand sides
% output:
%    x               array of solutions
%
% written by John Dannenhoffer
```

This header comment includes the following information:

- a brief description of what the function does
- a description of inputs (if any) including units, etc.
- a brief description of outputs that are produced
- the name of author of the function

Code formatting

Organize your code to make it easier to read. It sound trite, but a "good" code is one that both functions properly (i.e., properly tells the computer what to do) as well as is easy to read and hence debug.

To accomplish this, the following have been found to be useful:

- Line up code whenever possible. For example:

```
atri(ix) = 1 + 2 * Fo;
btri(ix) =       - Fo;
ctri(ix) =       - Fo;
dtri(ix) = T(ix,ntime);
```

- Use spaces around "big" operators and after commas.
- Use parentheses when precedence is not obvious; this is particularly useful in compound Boolean operations such as:

```
((a < b) || (a < c)) && (d > 0)
```

- Avoid long lines; anytime that you need to scroll left and right to read a line of code, it is too long. When you break up a line of code, use the ellipsis (...) operator, such as:

```
xform = [cos(theta), -sin(theta), 0; ...
         sin(theta),  cos(theta), 0; ...
         0,           0,          1];
```

- Use spaces between code "paragraphs", which is a set of lines that accomplish some sub-task.
- Use comments throughout code to define variables and to describe "what" is being done (not "how"). There is a tendency to skip this step with the intention of filling in the comments later. This is bad for two reasons:
  - despite good intentions, writers of code seldom spend the time to do this later
  - but more importantly, writing good comments when the code is being written makes it easier to debug and often leads to discovery of errors
- Always use MATLAB's suggested indentation, which is 4 spaces for `if/elseif/else/end` constructs and `for` and `while` loops.
- Always put a comment after an `end` statement so that the reader knows what is ended. For example:

```
for i = 1 : 10
    :
    if     (i == 5)
        :
    elseif (i == 6)
        :
    end % if
end % for i
```

- Put a separator line between functions and sub-functions, such as:

```
%--------------------------------------------------------------------
```

- Avoid putting more that one statement on a line, unless the statements are very tightly coupled..
- Put parentheses around the logical parts of `if` and `while` statements, such as:

```
if (a < b)
```

and

```
while (dist > 0)
```

Variable and function names

Define (variable and function) names carefully. Just using the first name that pops into your mind generally is not best. Specific guidelines that are used by many code authors include:

- Use names that are descriptive, but not too long. In general, variable names between 3 and 20 characters seem best.
- Use uppercase characters for first letter of each word in a name, such as:

  ```
  localPressure
  ```

- When dealing with collections (such as arrays), use variables that start with `i`, `j`, and `k` to index a particular member of the set (array) and variables that start with `m` and `n` to indicate the number of elements in the collection, such as:

  ```
  for icircle = 1 : ncircle
      area(icircle) = pi * radius(icircle).^2;
  end % for icircle
  ```

- Avoid using reserved words for names. The command:

  ```
  iskeyword
  ```

  can be used to check is a name is one of MATLAB's keywords. The current set of keywords include `break`, `case`, `catch`, `classdef`, `continue`, `else`, `elseif', \verbend‘, `for`, `function`, `global`, `if`, `otherwise`, `parfor`, `persistent`, `return`, `spmd`, `switch`, `try`, and `while`.

- Be consistent in the names that you choose. For example, do not use `r`, `R`, `rad`, and `radius` to mean the same thing.
- Avoid using `l` (the letter "el") as a variable name, since it is hard to distinguish from the number "1". Similarly, avoid using `O` (the uppercase letter "oh") since it is hard to distinguish from the number "0".

Other Good Habits

- Pre-allocate matrices (arrays) whenever possible. This is done with statements such as:

```
x = zeros(nx+1, 1      );
t = zeros(1,    ntime+1);
T = zeros(nx+1, ntime+1);
```

  or

```
circ = pi * ones(3, 5);
```

- Try to avoid using global variables. Every once in a while using global variables will make a code more readable, but in general it is good to avoid them for at least two reasons:
  - there is no protection for global variable, and so it is possible for two functions to use the same global variable name to refer to different items. If you must use a global variable, prefix the name with something (perhaps the name of the function) to avoid global name conflicts.
  - global variables are not automatically cleared when you enter a function, and hence a running program may start with "cruft" left over from the previous run

- Initialize loop results immediately before the loop, as in:

```
length = 0;
for i = 1 : 10
    length = length + width(i) * length(i);
end % for i
```

- When writing a fractional number (such as one-half), always put a digit before the decimal point (such as `0.5`) so that it is very obvious that the decimal point is part of the number and not a nearby operator (such as `.*`) or field separator.

- Avoid testing if two floating point values are exactly equal to each other; instead test if their values are sufficiently close to each other. For example,

```
if ((log(2) + log(2.5)) == log(5))
```

  will fail, but the test:

```
if (abs((log(2)+log(2.5)) - log(5)) < 1e-10)
```

  will pass.

Reference Guide

Operators and special characters.	
`+`	Plus; addition operator.
`-`	Minus; subtraction operator.
`*`	Scalar and matrix multiplication operator.
`.*`	Array (element-by-element) multiplication operator.
`^`	Scalar and matrix exponentiation operator.
`.^`	Array (element-by-element) exponentiation operator.
`\`	Left-division operator.
`/`	Right-division operator.
`.\`	Array (element-by-element) left-division operator.
`./`	Array (element-by-element) right-division operator.
`:`	Colon; generates regularly spaced elements and represents an entire row or column.
`( )`	Parentheses; encloses function arguments and array indices; overrides precedence.
`[ ]`	Brackets; encloses array elements.
`{ }`	Braces; encloses cell elements.
`.`	Decimal point.
`...`	Ellipsis; line-continuation operator.
`,`	Comma; separates statements, and elements in a row of an array.
`;`	Semicolon; separates columns in an array, and suppresses display.
`%`	Percent sign; designates a comment, and specifies formatting.
`'`	Quote sign and transpose operator.
`.'`	Non-conjugated transpose operator.
`=`	Assignment (replacement) operator.

Logical and relational operators.	
`==`	Relational operator; equal to.
`~=`	Relational operator, not equal to.
`<`	Relational operator, less than.
`<=`	Relational operator, less than or equal to.
`>`	Relational operator, greater than.
`>=`	Relational operator, greater than or equal to.
`&`	Logical operator, AND.
`|`	Logical operator, OR.
`~`	Logical operator, NOT.

Order of precedence.	
Highest	Parentheses, evaluated starting with the innermost pair.
	Transpose and exponentiation, evaluated left to right.
	Unary plus or minus and logical NOT ( ), evaluated left to right.
	Multiplication and division, evaluated from left to right.
	Addition and subtraction, evaluated from left to right.
	Colon operator (:).
	Relational operators, evaluated from left to right.
	Logical AND (&), evaluated left to right.
Lowest	Logical OR (—), evaluated left to right.

Special variables and constants.	
`ans`	Most recent answer.
`eps`	Accuracy of floating point precision.
`i,j`	The imaginary unit (square root of -1).
`Inf`	Infinity.
`NaN`	Undefined numerical result (not a number).
`pi`	The number π.

Commands for managing a session.	
`clc`	Clears Command window.
`clear`	Removes variables from memory.
`doc`	Displays documentation.
`exist`	Checks for existence of file or variable.
`global`	Declares variables to be global.
`help`	Displays help text in the Command window.
`helpwin`	Displays help text in the Help Browser.
`lookfor`	Searches help entries for a keyword.
`quit`	Stops MATLAB.
`who`	Lists current variables.
`whos`	Lists current variables (long display).

System and file commands.	
`cd`	Changes current directory.
`date`	Displays current date.
`delete`	Deletes a file.
`diary`	Switches on/off diary file recording.
`dir`	Lists all files in current directory.
`get`	Returns diary status or filename.
`load`	Loads workspace variables from a file.
`path`	Displays search path.
`pwd`	Displays current directory.
`save`	Saves workspace to a file.
`type`	Displays contents of a file.
`what`	Lists all MATLAB files.
`wk1read`	Reads .wk1 spreadsheet file.
`xlsread`	Reads .xls spreadsheet file.

Input/output commands.	
`disp`	Displays contents of an array or string.
`dlmwrite`	Writes formatted data to an ASCII file.
`format`	Controls screen-display format.
`fprintf`	Performs formatted writes to screen or file.
`input`	Displays prompts and waits for output.
`menu`	Displays a menu of choices
`;`	Suppresses screen printing.

Numeric display formats	
`format short`	Four decimal digits (default)
`format long`	16 decimal digits.
`format short e`	Five digits plus exponent.
`format long e`	16 digits plus exponents.
`format bank`	Two decimal digits.
`format +`	Positive, negative, or zero.
`format rat`	Rational approximation.
`format compact`	Suppresses some line feeds.
`format loose`	Resets to less compact display mode.

Array functions	
`cat`	Concatenates arrays.
`find`	Finds indices of nonzero elements.
`length`	Computes number of elements.
`linspace`	Creates regularly spaced vector.
`logspace`	Creates logarithmically spaced vector.
`max`	Returns largest element.
`min`	Returns smallest element.
`size`	Computes array size.
`sort`	Sorts each column.
`sum`	Sums each column.

Special matrices	
`eye`	Creates an identity matrix.
`ones`	Creates an array of ones.
`zeros`	Creates an array of zeros.

Matrix functions for solving linear equations	
`det`	Computes determinant of an array.
`inv`	Computes inverse of a matrix.
`pinv`	Computes pseudo-inverse of a matrix.
`rank`	Computes rank of a matrix.
`rref`	Computes reduced row echelon from.

Exponential and logarithmic functions.	
`exp(x)`	Exponential; e^x.
`log(x)`	Natural logarithm; $\ln(x)$.
`log10(x)`	Common (base ten) logarithm; $\log_{10}(x)$.
`sqrt(x)`	Square root of x.

Complex functions.	
`abs(x)`	Absolute value of x.
`angle(x)`	Angle of a complex number x.
`conj(x)`	Complex conjugate of x.
`imag(x)`	Imaginary part of a complex number x.
`real(x)`	Real part of a complex number x.

Numeric functions.	
`ceil`	Rounds to the nearest integer toward ?.
`fix`	Rounds to the nearest integer toward zero.
`floor`	Rounds to the nearest integer toward -?.
`round`	Rounds toward the nearest integer.
`sign(x)`	Returns +1, 0, or -1, depending on sign of x.

Trigonometric functions.	
`acos(x)`	Inverse cosine; $\cos^{-1}(x)$.
`acot(x)`	Inverse cotangent; $\sec^{-1}(x)$.
`acsc(x)`	Inverse cosecant; $\csc^{-1}(x)$.
`asec(x)`	Inverse secant; $\sec^{-1}(x)$.
`asin(x)`	Inverse sine; $\sin^{-1}(x)$.
`atan(x)`	Inverse tangent; $\tan^{-1}(x)$.
`atan2(y,x)`	Four-quadrant inverse tangent of y/x.
`cos(x)`	Cosine; $\cos(x)$.
`cot(x)`	Cotangent; $\cot(x)$.
`csc(x)`	Cosecant; $\csc(x)$.
`sec(x)`	Secant; $\sec(x)$.
`sin(x)`	Sine; $\sin(x)$.
`tan(x)`	Tangent; $\tan(x)$.

Hyperbolic functions.	
`acosh(x)`	Inverse hyperbolic cosine..
`acoth(x)`	Inverse hyperbolic cotangent.
`acsch(x)`	Inverse hyperbolic cosecant.
`asech(x)`	Inverse hyperbolic secant.
`asinh(x)`	Inverse hyperbolic sine.
`atanh(x)`	Inverse hyperbolic tangent.
`cosh(x)`	Hyperbolic cosine.
`coth(x)`	Hyperbolic cotangent.
`csch(x)`	Hyperbolic cosecant.
`sech(x)`	Hyperbolic secant.
`sinh(x)`	Hyperbolic sine.
`tanh(x)`	Hyperbolic tangent.

Polynomial functions.	
`conv`	Computes product (convolution) of two polynomials.
`deconv`	Computes ratio (deconvolution) of polynomials.
`eig`	Computes the eigenvalues of a matrix.
`poly`	Computes polynomial from roots.
`polyfit`	Fits a polynomial to data.
`polyval`	Evaluates polynomial.
`roots`	Computes polynomial roots.

String functions.	
`findstr`	Finds occurrences of a string.
`strcmp`	Compares strings.

Logical Functions.	
`any`	True if any elements are nonzero.
`all`	True if all elements are nonzero.
`find`	Finds indices of nonzero elements.
`finite`	True if elements are finite.
`isnan`	True if elements are undefined.
`isempty`	True if elements are infinite.
`isreal`	True if elements are real.
`xor`	Exclusive OR.

Miscellaneous mathematical functions.	
`cross`	Computes cross products.
`dot`	Computes dot products.
`function`	Creates a user-defined function.

Cell array functions.	
`cell`	Creates cell array.
`celldisp`	Displays cell array.
`cellplot`	Displays graphical representation of cell array.
`num2cell`	Converts numeric array to cell array.
`deal`	Matches input and output lists.
`iscell`	Identifies cell array.

Structure functions.	
`fieldnames`	Returns field names in a structured array.
`getfield`	Returns field contents of a structure array.
`isfield`	Identifies a structure field array.
`isstruct`	Identifies a structure array.
`rmfield`	Removes a field from a structure array.
`setfield`	Sets contents of a field.
`struct`	Creates structure array.

Basic xy plotting commands.	
`axis`	Sets axis limits.
`fplot`	Intelligent plotting of functions.
`ginput`	Reads coordinates of the cursor position.
`grid`	Displays gridlines.
`plot`	Generates x-y plot.
`print`	Prints plot or saves to a file.
`title`	Puts text at top of plot.
`xlabel`	Adds text label to x-axis.
`ylabel`	Adds text label to y-axis.

Plot-enhancement commands.	
`axes`	Creates axes objects.
`gtext`	Enables label placement by mouse.
`hold`	Freezes current plot.
`legend`	Legend placement by mouse.
`refresh`	Redraws current figure window.
`set`	Specifies properties of objects such as axes.
`subplot`	Creates plots in sub-windows.
`text`	Places string in figure.

Specialized plot functions.	
`bar`	Creates bar chart.
`loglog`	Creates log-log plot.
`plotyy`	Enables plotting on left and right axes.
`polar`	Creates polar plot.
`quiver`	Plot velocity vectors as arrows.
`semilogx`	Creates semilog plot (logarithmic abscissa).
`semilogy`	Creates semilog plot (logarithmic ordinate).
`stairs`	Creates stair plot.
`stem`	Creates stem plot.

Three-dimensional plotting functions.	
`contour`	Creates contour plot.
`mesh`	Creates 3D mesh surface plot.
`meshc`	Same as mesh with contour plot underneath.
`meshz`	Same as mesh with vertical lines underneath.
`plot3`	Creates 3D plots from lines and points.
`surf`	Creates shaded 3D mesh surface plot.
`surfc`	Same as surf with contour plot underneath.
`meshgrid`	Creates rectangular grid.
`waterfall`	Same as mesh with mesh lines in one direction.
`zlabel`	Adds text label to z-axis.

Program flow control.	
`break`	Terminates execution of a loop.
`case`	Provides alternate execution path within switch structure.
`continue`	Passes control to the next iteration of a for or while loop.
`else`	Delineates alternate block of statements.
`elseif`	Conditionally executes statements.
`end`	Terminates for, while, and if statements.
`for`	Repeats statements a specific number of times.
`if`	Executes statements conditionally.
`otherwise`	Provides optional control within a switch structure.
`switch`	Directs program execution by comparing input with case expressions.
`while`	Repeats statements an indefinite number of times.

Optimization and root-finding functions.	
`fminbnd`	Finds the minimum of a function of one variable.
`fminsearch`	Finds the minimum of a multivariable function.
`fminunc`	Finds the unconstrained minimum of a multivariable function
`fmincon`	Finds the minimum of a multivariable function with constraints.
`fzero`	Finds the zero of a function.
`optimset`	Creates optimization options structure.

Histogram functions.	
`bar`	Creates a bar chart.
`hist`	Aggregates the data into bins.

Statistical functions.	
`cumsum`	Computes the cumulative sum across a row.
`erf(x)`	Computes the error function, erf(x).
`mean`	Computes the mean.
`median`	Computes the median.
`std`	Computes the standard deviation.

Random number functions.	
`rand`	Generates uniformly distributed random numbers between 0 and 1; sets and retrieves the state.
`randn`	Generates normally distributed random numbers; sets and retrieves the state.
`randperm`	Generates random permutations of integers.

Polynomial functions.	
`poly`	Computes the coefficients of a polynomial and its roots.
`polyfit`	Fits a polynomial to data.
`polyval`	Evaluates a polynomial and generates error estimates.
`roots`	Computes the roots of a polynomial from its coefficients.

Interpolation functions.	
`interp1`	Linear and cubic-spline interpolation of a uni-variable function.
`interp2`	Linear interpolation of a function of two variables.
`spline`	Cubic-spline interpolation.
`unmkpp`	Computes the coefficients of cubic-spline polynomials.

Numerical differentiation and integration functions	
`diff(x)`	Computes the differences between adjacent elements in a vector x
`polyder`	Differentiates a polynomial, a polynomial product, or a polynomial quotient.
`quad`	Numerical integration with adaptive Simpson's rule.
`quad1`	Numerical integration with Lobatto quadrature.
`trapz`	Numerical integration with the trapezoidal rule.

ODE solvers.	
`ode23`	Nonstiff, low-order solver.
`ode45`	Nonstiff, medium-order solver.
`ode113`	Nonstiff, variable-order solver.
`ode23s`	Stiff, low-order solver.
`ode23t`	Moderately-stiff, trapezoidal rule solver.
`ode23tb`	Stiff, low-order solver.
`ode15s`	Stiff, variable-order solver.
`odeset`	Creates integrator options structure for ODE solvers.

Predefined input functions.	
`gensig`	Generates a periodic sine, square, or pulse input.
`sawtooth`	Generates a periodic sawtooth input.
`square`	Generates a square wave input.
`stepfun`	Generates a step function input.

Functions for creating and evaluating symbolic expressions.	
`class`	Returns the class of an expression.
`digits`	Sets the number of decimal digits used to do variable precision arithmetic.
`double`	Converts an expression to numeric form.
`ezplot`	Generates a plot of a symbolic expression.
`findsym`	Finds the symbolic variables in a symbolic expression.
`numden`	Returns the numerator and denominator of an expression.
`sym`	Creates a symbolic variable.
`syms`	Creates one or more symbolic variables.
`vpa`	Sets the number of digits used to evaluate expressions.

Functions for manipulating symbolic expressions.	
`collect`	Collects coefficients of like powers in an expression
`expand`	Expands an expression by carrying out powers.
`factor`	Factors an expression.
`poly2sym`	Converts a polynomial coefficient vector to a symbolic polynomial.
`pretty`	Displays an expression in a form that resembles typeset mathematics.
`simple`	Searches for the shortest form of an expression.
`simplify`	Simplifies an expression using Maple's simplification rules.
`subs`	Substitutes variables or expressions.
`sym2poly`	Converts an expression to a polynomial coefficient.

Symbolic solution of equations.	
`solve`	Solves symbolic equations.
`dsolve`	Returns the symbolic solution of a differential equation or set of equations.

Symbolic calculus functions.	
`diff`	Returns the derivative of an expression.
`dirac`	Dirac delta function (unit impulse).
`heaviside`	Heaviside function (unit step).
`int`	Returns the integral of an expression.
`limit`	Returns the limit of an expression.
`symsum`	Returns the symbolic summation of an expression.
`taylor`	Returns the Taylor series of a function.

Symbolic linear algebra functions.	
`det`	Returns the determinant of a matrix.
`eig`	Returns the eigenvalues (characteristic values) of a matrix.
`inv`	Returns the inverse of a matrix.
`poly`	Returns the characteristic polynomial of a matrix.

Laplace transform functions.	
`ilaplace`	Returns the inverse Laplace transform.
`laplace`	Returns the Laplace transform.

Sample Program (MatlabGuidelines.m)

```
function [] = MatlabGuidelines()
    % MatlabGuidelines - this program simulates unssteady 1d heat
    %    condduction with (unsteady) temperatures prescribed
    %     at the boundaries
    %
    % inputs:  (none)
    % outputs: (none)
    %
    % written by John Dannenhoffer

    % get all inputs
    xmax  = input('Enter xmax: ');              % maximum depth                    (m)
    nx    = input('Enter nx:   ');              % number of points in x direction (-)
    tmax  = input('Enter tmax: ');              % maximum time                     (d)
    ntime = input('Enter N:    ');              % maximum number of time steps     (-)

    alfa = 0.52 * 3600 * 24 / 2050 / 1840;      % diffusivity                      (m2/d)
    fprintf(1, 'Enter alfa: %f\n', alfa);

    dx   = xmax / nx;                           % distance between nodes           (m)
    dt   = tmax / ntime;                        % time step                        (d)

    Fo   = alfa * dt / (dx^2);                  % Fourier number

    % pre-allocate arrays
    x = zeros(nx+1, 1      );
    t = zeros(1,    ntime+1);
    T = zeros(nx+1, ntime+1);

    atri = zeros(nx+1 ,1);
    btri = zeros(nx+1, 1);
    ctri = zeros(nx+1, 1);
    dtri = zeros(nx+1, 1);

    % set up space array and initial temperatures
    for ix = 1 : nx
        itime = 1;
        x(ix      ) = (ix-1) * dx;              % spatial position                 (m)
        t(   itime) = 0;                        % initial time                     (d)
        T(ix,itime) = 20;                       % initial temperature              (C)
    end % for ix

    % step through remaining times
    for itime = 2 : ntime+1
```

```
    t(itime) = (itime-1) * dt;

    % set up the coefficients...

    % ...left boundary condition
    atri(1) = 1;
    btri(1) = 0;
    ctri(1) = 0;
    if (t(itime) < 60)                      % left boundary temperature       (C)
        dtri(1) = -15;
    else
        dtri(1) = +20;
    end % if

    % ...interior (governing equation)
    for ix = 2 : nx
        atri(ix) = 1 + 2 * Fo;
        btri(ix) =       - Fo;
        ctri(ix) =       - Fo;
        dtri(ix) = T(ix,ntime);
    end % for ix

    % ...right boundary condition
    atri(nx+1) = 1;
    btri(nx+1) = 0;
    ctri(nx+1) = 0;
    dtri(nx+1) = 20;                        % right boundary temperature     (C)

    % use the Thomas algorithm to solve the tridiagonal system
    Tnew = thomas(nx+1, atri, btri, ctri, dtri);

    % put the "results" into the temperature array
    for ix = 1 : nx+1
        T(ix,itime) = Tnew(ix);
    end % for ix
end % for itime

% find the minimum depth for which T > 0
mindepth = x(nx+1);

for ix = nx : -1 : 1
    okay = 1;

    for itime = 1 : ntime+1
        if (T(ix,itime) < 0)
            okay = 0;
```

```
            end % if
        end % for itime

        if (okay == 1)
            mindepth = x(ix);
        end % if
    end % for ix

    fprintf(1, '\nMinimum depth that pipe can be buried is approximately %f(m)\n', mindepth);

    % plot a contour of the results
    for itime = 1 : ntime+1
        for ix = 1 : nx+1
            tt(ix,itime) = t(itime);
            xx(ix,itime) = x(ix);
        end % for ix
    end % for itime

    [cs,h] = contourf(xx, tt, T); ...
        colorbar
        title('Contours of temperature (C)')
        xlabel('x (m)')
        ylabel('t (d)')

end % function MatlabGuidelines

%---------------------------------------------------------------------

function [x] = thomas(n, a, b, c, d);
    % thomas - solve a tridiagonal matrix using the Thomas algorithm
    %
    %  [a1  c1                  ] [x1]   [d1]
    %  [b2  a2  c2              ] [x2]   [d2]
    %  [    b3  a3  c3          ] [x3]   [d3]
    %  [                        ] [. ] = [. ]
    %  [             ...        ] [. ]   [. ]
    %  [                        ] [. ]   [. ]
    %  [                 bn  an ] [xn]   [dn]
    %
    % inputs:
    %    n               size of matrix
    %    a               array of      diagonal elements
    %    b               array of   subdiagonal elements
    %    c               array of superdiagonal elements
    %    d               array of right-hand sides
    % output:
```

```
    %    x               array of solutions

    % Forward elimination
    p(1) = - c(1) / a(1);
    q(1) =   d(1) / a(1);

    for i = 2 : n
        p(i) =  - b(i)                  / (a(i) + c(i) * p(i-1));
        q(i) = (  d(i) - c(i) * q(i-1)) / (a(i) + c(i) * p(i-1));
    end % for i

    % Back substitution
    x(n) = q(n);

    for i = n-1 : -1 : 1
        x(i) = p(i) * x(i+1) + q(i);
    end % for i

end % function thomas
```