MATLAB 7
Class Notes

Richard I. Shrager
Mathematical and Statistical
Consulting Laboratory
Center for Information Technology
National Institutes of Health
Building 12A, Room 2003C
Bethesda MD 20892-5620
Mail stop: 5620
Telephone: (301)402-5864
Fax: (301)402-4544
E-mail: shragerr@mail.nih.gov

September 21, 2006

Preface

This book is intended to serve as class notes for a six-hour course at the National Institutes of Health. The emphasis is on the core language of MATLAB, that is, data types, program elements, and syntax. Peripheral (though very important) issues like input/output, graphics, and user interfaces, are not covered to any significant extent. Those topics tend to require long lists of options which are awkward to fit into a brief course. The topics are chosen with two goals in mind: 1) avoiding initial confusions that previous students have encountered in their early use of MATLAB, and 2) highlighting especially useful computing techniques.

Contents

Chapter 1

Amenities

MATLAB is a high-level language for computing with numerical arrays and other data types. But before we get into that aspect of it, there are several features of the language that are not computing *per se*, but which do make your life easier.

1.1 Online Help

When you give a `help` or `doc` command, e.g. `help sqrt` or `doc sqrt`, MATLAB prints a description of the function `sqrt`, including what it does and how to use it. The `help` command displays the info directly in the command window, which can be annoying at times because it obscures what you were working on. By contrast, the `doc` command opens its own window. The `doc` entries are more instructive than the `help` entries. They are longer, more complete, and with more examples.

Accessing help files is somewhat cryptic, because the item you request must be a single name that the help command recognizes. (The doc files are not much better.) The help command will not accept a search criterion like a phrase, as a web browser might. To find your way around the help files, proceed systematically. You can start by saying `help`, which lists all primary (top-level) help topics, with a one-line explanation of each. For example, one of those primary topics is `elfun`, the category of elementary math functions. Then by typing `help elfun`, you get a listing of all functions within that category, along with a one-line description of each. For example, `sqrt` is one of those functions. Then by typing `help sqrt`, you get the needed details about that function. Of special interest is the final line of most entries, labeled "See also", which lists other entries you might want to use with or instead of this one. Thus, having found one entry of interest, you can let that entry guide you to others.

Each primary topic corresponds to a directory provided with MATLAB, or in some cases, written by you. By using the commands `help help` or `doc help`, you can learn about setting up help features for your own MATLAB programs. Taking advantage of such features can make life much easier for those using and maintaining your code.

Throughout these notes, special lines will appear beginning with the word `HELP:`. The topics on that line are used e.g. as in `doc sqrt` to get more information about the matters under discussion.

```
HELP:  help  doc
```

1.2 The Book

If you get serious about MATLAB programming, online help will not be enough. Online help provides information in separate packets organized under broad topics. It does not tie the language together very well. It does not show how the various features interrelate. For the best MATLAB 7 overview in print, buy this book:

Mastering MATLAB 7
by Duane Hanselman and Bruce Littlefield
Pearson Prentice Hall 2005
ISBN 0-13-143018-1

1.3 The People Option

It has been suggested that Moses and his people wandered the desert for 40 years because he was too stubborn to ask directions. If all else fails, you of course have the sense to ask. MATLAB is a product of:

The MathWorks, Inc.
3 Apple Hill Drive
Natick MA 01760-2098

whose various e-addresses are:

508-647-7000	Phone
508-647-7001	Fax
http://www.mathworks.com	Web site.
info@mathworks.com	Sales, pricing, general info.
support@mathworks.com	Technical support.
bugs@mathworks.com	Bug reports.
suggest@mathworks.com	Suggestions.
service@mathworks.com	Order status, license info.
doc@mathworks.com	Documentation error reports.

In my experience, the MathWorks support staff is prompt and helpful.

1.4 Start, Stop, and Emergency Stop

To start, go to your working directory, or be sure that your file-access path will let you get to the files you need. This is a system issue, and you should consult with your local computer staff, if any, for approved procedures. To enter MATLAB, click on the MATLAB icon, or whatever convention has been set up on your computer. Once in MATLAB, to find out which version you are running, and which toolboxes are available, type the command `ver`.

To get out of MATLAB, say `exit` or `quit`, or click the X in the upper right corner of the command window. Once in MATLAB, if you create a monster process that threatens to run forever or produce gobs of unwanted output, you can interrupt that process with control-c, which will return control of the session to you in most cases.

```
HELP:  ver  exit  quit
```

1.5 Creating a Diary

A diary is an ASCII file listing all the commands that you have issued during a session, along with MATLAB's printed responses to them. The command `diary on` starts the diary. You can say `diary off` at any time in your session, do things you don't want in the diary, then say `diary on` again when you want to resume. I prefer to leave the diary on during the session, then if necessary, edit the diary file after the session. That way, I don't forget to turn the diary back on again. Also, I'm never sure what might prove important during the heat of computation.

A diary is not limited to one session. You can say e.g. `diary MyDiary`, which will append the diary to the file MyDiary,which may contain a diary from past sessions. If a diary file name is not specified, the file name `diary` will be used.

Forgetting to start or restart a diary in an otherwise productive session can be most frustrating. To insure that your diary will be on at the outset, put a `diary on` statement in your startup.m file, which MATLAB calls upon entry, so you can't forget it. The diary will be saved upon exit.

```
HELP:  diary
```

1.6 Utility Windows

In a MATLAB session, several windows are available:

- The Command window shows what you are doing now.
- The History window shows what you have done recently.
- The Directory window shows file names in the current directory.
- The Edit window shows the text of the selected file you are editing.
- The Graphics window shows plots and images.
- The Profile window shows the computer time consumed by each file during a time trial.
- The `doc` window shows detailed documentation about a user-selected feature in HTML-style format.

- The Debug window becomes active whenever a “break” point is reached in the program. You choose the break points using debug commands. There are also special debug commands, e.g. `dbstop if error` will stop where an error occurs, so you know where it happened and can display the values that caused it.

```
HELP:  dir  pwd  cd  edit  plot  profile  doc
       dbcont  dbstep  dbclear  dbtype  dbstack
       dbup  dbdown  dbstatus  dbquit
```

1.7 User-Defined Interfaces

MATLAB supports graphical interfaces defined by the user, such as pushbuttons, wait bars, pull-down menus, and the like.

```
HELP:  uitools
```

1.8 Functions and Scripts

In general, commands to MATLAB may be issued directly by you, or by M-files (files named with the suffix `.m`) that you invoke. There are two styles of M-file. The simplest is the script file. When you invoke a script by typing its name without the `.m` extension, the statements it issues are indistinguishable from the statements you issue from the keyboard. Variables defined by you or that script file are available to you and the script file. This convention requires some care in choosing variable names, to avoid inadvertent altering of essential values. By contrast, function files communicate (mostly) through input and output arguments. Variables within a function are private to that function, and to the script files that it invokes. (Scripts are fickle in that they use the variables of any process that invokes them.) To illustrate, let’s look at a session that uses a script file and a function.

```
% ----- This comment begins the function file hyperb.m -----
function Y=hyperb(X)
X=2*X;   % This X has no bearing on X in the master session.
Y=1 ./ (1+X);   % Same idea for Y.
% ----- This comment ends the function file hyperb.m ------

% ----- This comment begins the script file dubble.m, -------
% ----- which is invoked by the master session.
X=2*X;   % Master session's X is doubled.
% ----- This comment ends the script file double.m ----------

% ----- This comment begins the master session -----
X = [1,1,2,3,5,8];    Y = X+5;
Z = hyperb(Y);   % This statement invokes the function file.
                 % Master session X & Y remain unchanged.
```

```
                     % Only Z is affected.
    dubble;          % This statement invokes the script file.
                     % Master session X is now [2,2,4,6,10,16];
```

1.9 Statement delimiters

MATLAB accepts executable statements from the keyboard, or from a file, in effect one line at a time. To continue a statement over more than one line, every non-final line of the statement must end in an ellipsis (`...`). For example, here is a legal statement:

```
y = ( x^2 + 5*x + 6 ) ...
  / ( x^2 + 6*x + 7 );
```

but if the ellipsis is omitted, line 1 is accepted as legal while line 2 is regarded as a separate but malformed statement, producing an error. The ellipsis is important because some statements will not fit on a line, and because some multi-line statements are easier to read. All text to the right of an ellipsis is ignored.

Statements can also be ended with comma (`,`) or semicolon (`;`), enabling the placement of more than one statement on a line. Ending a statement with `;` also suppresses those mostly unwanted printouts of results. Multi-statement lines should be used sparingly. Indiscriminate jumbling of statements can destroy a program's readability.

1.10 Blanks and comments

Other features that improve readability are the blank and the comment. Blanks may be inserted anywhere except in the middle of a name (e.g. `delta`, but not `del ta`) or operator (e.g. `a .* b`, but not `a. *b`), or number (e.g. `1e6`, not `1 e6`). For further limitations, see the chapter on tricky stuff.

The character `%` anywhere on a line tells MATLAB to ignore the rest of the line, including other `%`'s. Thus the above example can be commented as follows:

```
% -- Compute the %!@#$&* rational function --
y = ( x^2 + 5*x + 6 ) ...    % Numerator.
  / ( x^2 + 6*x + 7 ) ;      % Denominator.
```

Notice that whole or partial lines can be comments, but an executable statement can never appear to the right of a comment.

Comments can also appear in block form, beginning with `%{` and ending, perhaps several lines later, with `%}`. Everything in between is comment, ignored as code. This provides an easy way e.g. to temporarily remove code that might be causing a problem.

1.11 Temporary Exits

If you cannot complete a session in one sitting, you can save all of your variables with a simple `save` command, and retrieve them later with the `load` command. For most versions of MATLAB, there is an efficient way to do non-MATLAB chores within a MATLAB session. Any command that begins with `!` is not executed by MATLAB, but is instead referred to the operating system:

```
!copy c:\olddirectory\MyFile MyFile
```

The above command will be executed by the system, not by MATLAB, and a copy of the file MyFile.m be placed in the current directory. When `copy` is done, MATLAB resumes with the session intact. Commands beginning with `!` can do anything understood by your system.

```
HELP:  load  save
```

1.12 Retrieving and Correcting Commands

As you are typing a command, mistakes can be corrected by backspacing etc. in the usual manner. Mistakes in a previous command can be corrected easily by hitting the “up-arrow” key, which recalls the previously-entered line for editing and/or reentry. Likewise, by hitting “up-arrow” repeatedly, any previously-entered line can be recalled. As a shortcut, type the first few characters of the line you want to retrieve, then “up-arrow” will access only lines that begin with those characters. As a further aid, MATLAB keeps a History window from which you can retrieve past commands.

1.13 Summary

To enable and ease your MATLAB experience, you can:

- start a MATLAB session,
- exit from a MATLAB session,
- stop a runaway process,
- create or continue a diary,
- access several useful windows,
- design user interfaces,
- get on-line help using `help` or `doc`,
- issue statements directly, or from M-files,
- exit temporarily and resume,
- suppress printout using semicolon,
- put more than one statement on a line,

- spread one statement over several lines,
- put comments in your code,
- insert blanks in most reasonable places,
- recall previous commands for editing and reuse.

Chapter 2

Data and Variables

The primary types of data in MATLAB are numbers, characters, logicals, and empty. The secondary types or containers of data are arrays, cells, structures, and objects. Of the container types, the array is central. Every variable in MATLAB is an array of some sort. An array may contain primary data or other containers which in turn may contain other containers and so on. The interrelations can get quite complicated, but they usually don't.

2.1 Numbers in General

Numbers are displayed as integers, or as decimal constants in either fixed point or floating point format. For example, the following are all representations of the number 132:

```
132  132.0  1.32e2  1.32e+2  132e0
```

Note that a sign may appear within a constant, but only immediately following `e` in the floating point form. A real constant may not contain blanks. In complex constants, the letter `i` or `j` is suffixed to the imaginary part, as in `2+3i` or `3e7 + 2e6j`. Blanks may appear between terms of an imaginary constant. As separate names, `i` and `j` represent $\sqrt{-1}$ unless redefined by the user. So we see that in addition to numerals, the symbols `+-.eij` may appear in numerical constants. There is also a hexadecimal (base 16, hex for short) format, which allows one to determine the exact value of a floating point number to the very last bit, but this is specialized topic, and will not be elaborated here.

2.2 Floating Point Numbers

Floating point numbers (FPNs) are most prevalent because of their mostly consistent precision and superior range. A FPN consists of two parts: a signed exponent and a signed mantissa. For example, consider a FP decimal system that allows all 5-place mantissas and all 2-place exponents. Thus `1.2345e-16` is a system value as is `1.2346e-16`. The latter number is the upper neighbor of the former number in the system. The epsilon function is defined as the difference between a FPN and its upper neighbor, as in

```
>>eps(1.2345e-16)  % You type this.
  ans = 1e-20      % The system responds.
```

There is no system value between neighbors. Of course, there are real numbers between them, but our chosen FP system excludes them. Other values are also excluded because the range of FPNs, while large, is finite. Numbers less than `-9.9999e99` or greater than `9.9999e99` are not in the system. If you try to produce a FPN outside the system range, an overflow error will occur.

Similarly, numbers of magnitude less than `0.0001e-99` are not in the system. If you try to produce a magnitude less than the system minimum, and underflow will occur, producing zero (in this case, `0.0000e00`).

MATLAB adheres to the IEEE floating point standard, in which there are single and double precisions, with the following system parameters:

```
                  Single Precision     Double Precision
Max Magnitude:    3.4028e+038           1.7977e+308
Min Magnitude:    1.1755e-038           2.2251e-308
eps(1e0):         1.1921e-007           2.2204e-016
```

Double precision offers much smaller epsilons, i.e., much closer neighbors, and much larger range. The down side of double is that a single-precision number takes up four bytes, while double precision takes eight. (Oddly, there is no time penalty for double, since the same registers are used for both precisions.) Now there are always some smarties in the class who say, “Who needs double precision? Single precision is precise to seven places. My data are only accurate to, say, four places.” Don’t believe these people even if they are you! Some error is made on most floating point operations. Modern computers perform millions of such operations a second. That’s millions of chances to get into numerical trouble. Double precision is not a cure-all, but it is worth every byte.

FP formats also provide some ill-defined values, like NaN (not a number), produced e.g. by 0/0, or inf (infinity), produced e.g. by 1/0.

```
HELP: double single eps realmax realmin isfloat
      round floor ceil fix isnumeric isfloat
      nan isnan inf isinf isfinite isprime
      isequal isequalwithequalnans
```

2.3 Integers

In addition to FP single and double precision, MATLAB has signed (`int`) and unsigned (`uint`) integers in 8-, 16-, 32- and 64-bit formats, eight integer types in all. These integers can have values in the following ranges:

Integer Type	Minimum Value	Maximum Value
uint8	0	$2^8 - 1$
uint16	0	$2^{16} - 1$
uint32	0	$2^{32} - 1$
uint64	0	$2^{64} - 1$
int8	-2^7	$2^7 - 1$
int16	-2^{15}	$2^{15} - 1$
int32	-2^{31}	$2^{31} - 1$
int64	-2^{63}	$2^{63} - 1$

Like FP data, arrays in these other formats may be real, or complex. In 8, 16, and 32 bit formats, addition, subtraction, and multiplication are done directly on the integers in the expected way. However, division with integers is done by converting to FPNs using the `double` function, dividing, then converting back. Especially with the shorter formats, some care may be required to insure that the answers lie in the appropriate ranges listed above.

The uint8 format is useful for compact image representation, especially in three-color format. This compactness is welcome because images are notorious for their sheer bulk in computer memory and disk space.

```
HELP: int8 int16 int32 int64 uint8 uint16 uint32 uint64
      colormap image imagesc infinfo imread imwrite
      intmax intmin isinteger
```

2.4 Characters and Strings

Character or string constants are enclosed in single quotes, such as:

```
'This is a string.'
```

Each character in a string is coded as a uint16 integer, but let's not dwell on that detail. Certain strings e.g. `'2.3e5'` represent numbers and can be converted to them. Inversely, numbers can be converted to the strings that represent them. However, string data has properties and privileges not shared by numbers. These will be discussed in the chapters on strings and cells.

```
HELP: char cellstr cell strvcat strfun
      ischar iscellstr isspace isletter
```

2.5 Logical Values

A logical constant is denoted 1 (true) or 0 (false), and is usually generated by comparisons like `a<b`, or by logical operations like `a&b`. In these notes, we will sometimes use the convention `T` for `true` and `F` for `false` to avoid the confusion with the integers 1 and 0. Numbers (other than complex) can be converted to logical using the `logical` function. For any number `x`

of any type except complex, `logical(x)` produces `false` when `x=0` and `true` otherwise. However, a warning will be given if x is not `0` or `1`. To avoid the warning, `x` can be converted to logical in an equivalent manner by the comparison `x~=0`. Comparisons between numbers always produces logicals. In the table below, `x` and `y` are numbers.

Comparison:	Produces `true` if and only if:
`x <  y`	x is less than y.
`x <= y`	x is less than or equal to y.
`x == y`	x equals y.
`x ~= y`	x does not equal y.
`x >= y`	x is greater than or equal to y
`x >  y`	x is greater than y.

In addition to comparisons, which accept numbers and produce logicals, there are purely logical operations which accept logicals and produce other logicals. The comparisons in the above table accept logical operands, in effect using the numerical values `0` for `false` and `1` for `true`. In the table below, let L1 and L2 be logical operands.

Operation:	Produces `true` if and only if:
`~L1`	L1 is false.
`and(L1,L2)`	Both L1 and L2 are true.
`L1 | L2`	Either or both are true.
`xor(L1,L2)`	Either but not both are true.

```
HELP: logical islogical and or xor not
```

2.6 Empty Arrays

The empty array, denoted `[ ]`, is MATLAB's place-keeper. It occupies an array that exists, but that has no data yet, or that has had all data removed. For example, all global and persistent variables are empty at the outset. An array can have several dimensions, but if any one of them is zero, the array is empty.

```
HELP: isempty
```

2.7 Names of variables

The naming conventions in MATLAB are similar to naming conventions other languages. A name may contain up to 63 characters chosen from upper and lower case alphabetic characters, decimal digits, and the underline character. No spaces, operators, or punctuation are permitted. (MATLAB allows more than 63 characters in a name, but only the first 63 will be recognized.) The first character must be alphabetic. Some examples are:

```
A  a  Jar23  n7Z  group_5_age  LaTeX_
```

Note that only the underline can serve as a separator for readability within a name, since spaces and punctuation within names are not allowed. Names are case-sensitive, so that the first two names above are distinct.

MATLAB key words may not be used as variable names. These words are:

```
function return    for while break continue
if else elseif     switch case otherwise
try catch          global persistent          end
```

If you forget the list, type `iskeyword` for the complete list, or if you wonder about a specific name, e.g. `bagel`, type `iskeyword('bagel')`, which will display `1` if `bagel` is a keyword, and `0` if it is not. However, keywords do not use upper case. Therefore, names like `IF` and `enD` may be used without conflict.

```
HELP:  iskeyword
```

2.8 Array variables

An array is a rectangular arrangement of data in rows, columns, pages, books, etc., so that every row contains the same number of columns, every page contains the same number of rows, every book contains the same number of pages, etc. Every variable in MATLAB has a name and contains an array of data. An array A that contains several rows, columns, pages, and books is a four-dimensional (4-D) array. If A contains only one page in only one book, and so on, it is a matrix. An empty array has at least one zero dimension. MATLAB assigns at least two dimensions to every array. Thus a single row or column, or a single number, or an empty array all have two dimensions, but the values of some of those dimensions will be 1's or 0's.

2.9 Manipulating Arrays

The five basic infix operations (`+,-,*,/,^`) and many functions (e.g., `log` and `sin`) are available for doing scalar arithmetic, which looks much the same as in FORTRAN or C.

```
a=exp(b)*sin(c)^log(d);   % a,b,c,d are scalars.
```

Elements of matrices can be treated as scalars by subscripting.

```
X(2,2)=exp(X(1,1))*sin(X(1,2))^log(X(2,1));
```

But the real value of MATLAB is its ability to treat a whole array as a notational unit, thus doing away with much of the looping found in lower-level languages. An array can be created explicitly, using square brackets. Elements of a row are delimited by commas or spaces, and rows are delimited by semicolons or end-of-lines.

```
X=[ 1,2,3; 4 5 6 ];   % X has 2 rows, 3 columns.
Y=[ 1 2 3             % Y is the
    4,5,6 ];          %    same as X.
```

Square brackets can also catenate matrices to form larger matrices.

```
Y=[ X,X; X,-X ];   % Using X from the above example,
                   % Y becomes [ 1 2 3   1  2  3
                   %             4 5 6   4  5  6
                   %             1 2 3  -1 -2 -3
                   %             4 5 6  -4 -5 -6 ]
```

Higher dimensional arrays can be created using the `cat` function:

```
A3 = cat(3,Y,Y,Y,Y);
```

In the above example, the array `Y` is catenated to itself in the third dimension, creating the 3-D array `A3` having four identical pages.

Matrices of arbitrary size can be generated, limited only by the workspace available.

```
u=0:.2:6;   % u is a row vector of values 0 to 6 in steps of 0.2.
Y=zeros(size(X));   % Y is a matrix of zeros, same size as X.
Z=randn(m,n);   % Z is an m-by-n matrix of random numbers.
```

Matrices can also be read from files. The following is a code for reading an ASCII file of numbers with arbitrary format into a vector v.

```
fid=fopen('matfile');   % Open the file matfile.
v=fscanf(fid,'%g');   % Read all nos. from matfile into vector v.
status=fclose(fid);   % Close matfile (which remains unchanged).
```

Parts of matrices can be extracted or deleted. Here is code for extracting, then deleting rows 2, 3, and 5 from an existing 6-by-6 matrix X:

```
v=[2,3,5];    % v is a vector of indexes.
Y=X(v,:);     % Y gets X rows 2, 3, & 5.   X is unchanged.
X(v,:)=[ ];   % X is now 3-by-6, with rows 2, 3, & 5 deleted.
```

All manner of operations can be performed with pre-existing matrices.

```
% ----- Let a,b,c be scalars.
% ----- Let r,s.t.u,v be vectors of appropriate size.
% ----- Let W,X,Y,Z be matrices of appropriate size.
W=sin(X);   % Every element W(i,j)=sin(X(i,j)).
Z=X*Y;      % Matrix product.
Z=X.*Y      % Array product: every Z(i,j)=X(i,j)*Y(i,j).
Y=X^2       % Y gets X*X, matrix 2nd power.
Y=X.^2      % Every Y(i,j)=X(i,j)^2.
% ----- In the following 3 statements, Y gets X inverse
% ----- when X is square and nonsingular.
Y=eye(size(X))/X;   Y=inv(X);   Y=X^(-1);
% ----- Functions can accept and produce any combination
% ----- of scalars, vectors, and matrices.
```

```
[X,Y,Z]=svd(W);   % W is factored into special matrices X.Y,Z.
% ----- Linear least-squares is easy in MATLAB.
u=X\v;   % u gets the least-squares solution to Xu=v.
s=X*u;   % s is the approximating function to v.
r=v-s;   % r is the residual vector for the above solution.
b=sum(r.*r);   % b is the sum of squares for the above solution.
% ----- If t is the independent variable vector for the above
% ----- problem, the following plots may be of use.
plot(t,r); % Plot the residuals r vs. t.
semilogy(t,v,t,s); % Compare data v & model s on a semilog plot.
```

2.10 Summary of array operations

Numerical arrays are generalizations of matrices, in that they can have an arbitrary number of indexes.

1. Arrays can operated upon in one-element-at-a-time style, i.e., incremented, decremented, multiplied, divided or exponentiated by a scalar (scalar-by-matrix operations), or by corresponding elements of another matrix (array operations).
2. In similar style, scalar functions like `sin` and `log` can be applied to arrays in one step, producing an element-for-element result.
3. One must be careful to distinguish between the above array-style operations and matrix operations like matrix product, matrix inverse, and matrix exponentiation, which are not element-for-element.
4. Arrays can be created using file-reading statements, array-initializing functions like `zeros`, colon notation for equally-spaced vectors, and square-bracket notation for arbitrary matrices.
5. Arrays can be built from smaller arrays using catenation, i.e. square brackets and the `cat` function.
6. Arrays can be rearranged using the transpose operator ('), various functions such as `reshape` and `flipud`, and indexing such as `v([3,2,1])=v(1:3)`, which reverses the order of the first 3 elements of `v`.
7. An array can be augmented by storing numbers in non-existing elements, as in `v(7)=0` where `v` had only five elements initially.
8. Cross-sections of arrays can be removed by storing empty information in them, e.g. so that `X(:,3)=[]` removes the third column of `X` (the fourth column becomes the third column and so on).

```
HELP: cat linspace logspace fliplr flipud flipdim
      rot90 permute transpose ctranspose reshape
      zeros ones rand sum prod
```

2.11 Numerical arrays

Numerical arrays may be scalar (1-by-1), vector (n-by-1 or 1-by-n), matrix (n1-by-n2), or of higher dimension (n1-by-n2-by-...). Empty arrays have at least one 0 dimension. For example, the statement `A(1,2,3,4)=5;` stores the value `5` in the first row, second column, third page, fourth book, if you will, of the array `A`. Indexing rules for matrices and arrays are discussed in the next chapter of these notes.

```
HELP:  double uint8 uint16 uint32 uint64
       single  int8  int16  int32  int64
       isfloat  isinteger
```

2.12 Sparse matrices

Sparse matrices are specialized numerical arrays. For applications where large matrices contain mostly zeros, the sparse data type may be of use. In a sparse matrix (dimensions beyond the second are not allowed for sparse data), only the nonzero elements are stored, along with their indexes. This may save an immense amount of storage, and may also permit faster processing in fortunate cases. Like the full (non-sparse) storage format, sparse matrices are stored in double precision, and may also be logical or complex (see below).

While sparse format can save workspace, MATLAB must store indexes along with nonzero entries, so each nonzero takes more space (16 bytes per double, 24 bytes per complex). For example, if a real matrix is half zeros, you save no space by using sparse format. Also, sparse indexing and many sparse numerical methods can have greater computational overhead (e.g. greater access time per element) than their full matrix counterparts. Often, there is a crossover in the efficiency curves of full and sparse methods, depending on array size and fraction of zeros. Sometimes, the need for sparse format is overwhelming, but if not, some experimentation may be in order to find the best format for your case. See the chapter on time and work.

```
HELP:  issparse  sparse  spalloc  spones
       speye  spconvert  full  find  sparfun
```

2.13 Logical arrays

Logical arrays are used with great dexterity and compactness in MATLAB, not only for controlling program flow in `if` statements and while-loops, but also for indexing and loop avoidance. The relational operators `( <, <=, ==, ~=, >=, > )` work in array fashion, as do the logical operators and functions `~, &, |, nor`. In an expression like `a>X` where `a` is scalar and `X` is a matrix, the result is a logical `0-1` matrix the same size as `X`, which contains a `1` wherever `a>X(i,j)` holds, and `0` otherwise. Alternately, `Y>X`, where `X` and `Y` are arrays of the same size, produces a logical array of the same size containing a `1` wherever `Y(i,j)>X(i,j)`, and `0` otherwise. These logical matrices can in turn be used as indexes to access or alter pertinent parts of a matrix, e.g.: `v=u(u>5)` copies the elements of vector `u` which are greater than 5 into the vector `v`. Logical indexing will be discussed in detail in the next chapter.

Logical variables can also be created from numerical variables by a simple conversion function: `L=logical(A)`. The logical variable `L` has the same size as `A`. The values in `L` will be `F` where `A` is `0`, and `T` otherwise. Logical data is stored one value to a byte.

```
HELP:  islogical logical and or xor not
```

2.14 Complex arrays

Complex arrays are a subclass of numerical arrays. A complex number is stored as a pair of double precision numbers, namely, the real and imaginary part, occupying 16 bytes per number (24 bytes per number in sparse format). Operations on real and complex data look alike, which can be disconcerting at times. For example, `sqrt(x)` will work even for `x<0`, producing an imaginary number when you might have preferred an error message. If an array is real (no imaginary parts),and you store a complex value in just one of its elements, the entire array is converted to complex format. MATLAB makes some effort to convert back to real format when all imaginary parts become zero. There are many MATLAB functions that are valid for both real and complex data, such as elementary functions like `sin` and `exp`, special math functions like the Bessel functions, and matrix operations like `inv` and `eig`.

```
HELP: isreal real imag conj angle abs
```

2.15 Bits

There is not really a separate data type for bits, but there are several MATLAB-provided functions that treat numerical data in bitwise fashion, enabling you to store, retrieve, and alter bits as though there were such a data type. These functions operate on unsigned integers, allowing up to 64 bits per number.

```
HELP: bitand bitcmp bitor bitmax
      bitxor bitset bitget bitshift
```

2.16 Cell and Structure Arrays

Cell and structure arrays may hold data of nonuniform type and size, including other cells and structures. The variety of possible data organization using such arrays is truly bewildering. A later chapter on cells and structures will provide enough detail to get you started.

```
Help: cell cell2struct celldisp cellplot deal num2cell
      fieldnames getfield rmfield setfield struct
      struct2cell iscell isfield isstruct
```

2.17 Function Handles

Functions are often asked to operate on other functions. Sometimes the function to be operated on varies from call to call. To ease the task of passing information about functions to other functions, the function handle was invented. A function handle is denoted by prefixing the function name with an `@` sign, as in `fh=@cos`. Function handles are strictly scalar, i.e. no function handle arrays or indexing. More detail is provided in the chapter on m-files.

```
HELP: feval func2str function_handle functions str2func
```

2.18 Classes and objects

Objects are structures with special rules of manipulation that depend on the class to which the objects belong. There are several built-in classes in MATLAB, e.g.: `double`, `sparse`, `char`, `struct`, and `cell`. Rules for your own classes are user-defined by files stored in specially-named directories. For example, the directory `@myobj` contains rules for objects of class `myobj`. Within that directory is a special file named `myobj.m` (same as the directory but without the `@` sign) known as a constructor. The constructor's job is to define new objects of the class `myobj`. Other files in the `@myobj` directory define the rules of manipulating objects of class `myobj`, including conversion to objects of other classes. Objects of a "child" class may inherit attributes from several "parent" classes, so that methods from a parent class can operate on appropriate parts of the "child" object. The rules can redefine operators like `+` and `*` so that something nonstandard happens to `A+B` when either `A` or `B` is an object.

```
HELP: class struct methods isa
      isobject  inferiorto  superiorto
```

Chapter 3

Grammar

3.1 The Assignment Statement

An assignment statement is of the form:

```
variable = expression
```

Such a statement directs MATLAB to evaluate the expression and store it in the variable. A variable can be of any type discussed in the previous chapter.

3.2 The Command

Commands are of the form `command arg1 arg2` with `arg1` and `arg2` being arguments. There can be as many arguments as needed. For example, in the command:

```
save X_Files FBI CIA MI5 ETs
```

the variables `FBI, CIA,` etc. are saved in a file called `X_Files`.

MATLAB has function-command duality, i.e. there are two ways to write a command. The simplest is in command form as shown above. But sometimes a command can also return a value, in which case fumction form is required. (See below and later chapter about functions.) For example, the `load` command retrieves what the `save` command saves:

```
load X_Files     Q = load('X_Files');
```

The statement on the left will simply restore the values `FBI, CIA`, etc., while the statement on the right will create a structure containing the input variables. (See below and later chapter about structures.) Each command and function operates differently, so it pays to consult the help files.

```
HELP: load save
```

3.3 The for-loop

Matrix conventions avoid many loops, but not all loops. A generic MATLAB for-loop looks like this:

```
for c = X;
   % Repeat this code for c = each column of X.
end;
```

If `X` is empty, the for-loop is bypassed. If `X` is a row n-vector (the most common case), the scalar `c` will be assigned each of the n elements of `X` in turn, and the for-loop will be executed n times unless interrupted (see below). If `X` is an m-by-n matrix with $m > 1$, the vector `c` will be assigned the columns of `X` in turn, and as above, the for-loop will be executed n times. Note that if `X` is a single column, the loop is executed only once, with `c=X`.

When the statement `for c=X` is first met, a list is made of the values (columns of `X`) that the looping variable `c` will become in each iteration. At the start of each ith loop, `c` becomes the ith item in the list, even if you have reset `c` or `X` during previous loops. Therefore, you cannot control the number of loops by resetting the looping variables.

A `continue` statement met within the loop causes the innermost loop that contains it to proceed immediately to the next iteration (if any), with the next value of `c`. You can get out of a for-loop before all iterations are completed, by using a `break` statement or a `return` statement within the loop. As with `continue`, a `break` statement applies only to the innermost loop that contains it. The `return` statement gets out of the function and returns to the program that called it.

```
HELP: for end break continue return
```

3.4 The while-loop

The while-loop is iterated as long as a given logical scalar expression is true, i.e.:

```
while L1;
   % Repeat as long as L1 is true.
end
```

In a while-loop, the entire test expression `L1` is re-evaluated using current values. Compare the two cases:

```
for a=b     while a=b
```

Altering `a` or `b` within the while-loop will alter the sequence of stopping tests, whereas altering them within the for-loop will not affect the sequence of `a` generated from `b`.

The while-loop has an inherent risk. Unlike the for-loop, there is no built-in guarantee that a while-loop will terminate, i.e. infinite while-loops are possible. It is up to the programmer to ensure that a stopping condition will occur. The `break`, `return`, and `continue` statements in `while` loops have the same effects as in `for` loops.

```
HELP:  while end break continue return
```

3.5 The if-elseif-else Construct

The `if` statement is similar to that in FORTRAN, e.g. let `L1`, `L2`, `L3`, etc. be logical scalar expressions:

```
if L1, y=f1(x); % This section is used if L1 is true.
elseif L2, y=f2(x); % Used if L1 is false and L2 is true.
elseif L3, y=f3(x); % if L1 and L2 are false and L3 is true.
     ... more elseif's if needed ...
else y=f4(x); % Used if L1, L2, L3, etc. are false.
end;
```

Only the `if` block is required. The `elseif` and `else` blocks are optional.

```
HELP:  if elseif else end
```

3.6 The switch-case Construct

The `switch` statement allows a multi-branched decision in concise form:

```
switch e0;   % e0 is some expression.
   case {a1,a2};      y=fa(x);   % used if e0=a1 or a2.
   case {b1,b2,b3};   y=fb(x);   % used if e0=b1, b2, or b3.
   case c1;           y=fc(x);   % used if e0=c1.
   % More cases if needed; otherwise (below) is optional.
   otherwise;         y=f(x);    % used if no match for e0.
end;
```

At most one case will be executed, after which control passes to the statement after `end`. If there is no match for e0, and the `otherwise` clause has been omitted, then the `switch` construct is simply bypassed. The `switch` clause is the only required clause. The `case` and `otherwise` clauses are optional. However, it is pointless to use the `switch` construct without `case` clauses, and it is often risky to omit the `otherwise` clause.

```
HELP:  switch case otherwise end
```

3.7 The try-catch Construct

Ordinarily, when a detectable error occurs, execution is stopped, an error message is printed, and control is returned to the user. However, when errors are anticipated in specific areas of the code, the user can code special reactions to them. The construct for automatic error trapping is:

```
try;
   % Error-prone block of code goes here.
   % If an error occurs,...
```

```
catch;
   % ... control is transferred here.  This is
   % a block of code that reacts to the error.
end;
```

A string called `lasterr` receives the error message, execution of the `try` block is discontinued at the point where the error occurred, and control is transferred to the `catch` block, which can use the string `lasterr` to figure out what to do. You can nest try-catches just as you can nest if-elseif-elses. When an error is trapped by a `catch` block, the error message is not printed unless there is an explicit statement to do so. Currently, the `catch` block is optional, which is dangerous. If you code a `try` block without a `catch` block, an error can occur with neither an error message nor a corrective action.

```
HELP: try catch lasterr end
```

3.8 Spaghetti Code

Consider the following description of a program:

```
STEP 0: Initialize variables. K=1.
STEP 1: Do Block 1, then go to Step K.
STEP 2: Do Block 2, then go to Step K.
STEP 3: Do Block 3, then go to Step K.
STEP 4: Final calculations and output.
```

Blocks 1, 2, and 3 are blocks of code in which the variable K may or may not be altered. From this outline, it is impossible to anticipate the order of the various steps, or indeed to tell if the program will ever stop. Such a description is called spaghetti code, because its path is conceptually a complete tangle. To avoid this kind of programming style, top-down programming was developed, in which there is no go-to statement. Well and good, but sometimes you must cope with a program description of the above type, which you can code in MATLAB, go-to or no:

```
K=1;           % Initialize K.
while true;       % Potentially infinite loop.
   switch K;         % Choose case K.
      case 1; Block1;   % Block1 is a script.
      case 2; Block2;   % Block2 is a script.
      case 3; Block3;   % Block3 is a script.
      case 4; break;    % Get out of while-loop.
   end;              % End switch.
end;              % End while. Do another loop.
Block4;        % Block4 is the final script.
```

Chapter 4

Indexing Arrays

This chapter discusses the process of indexing arrays, which is basic to the operation of the MATLAB language. In the early sections, we will consider cases where only one or two indexes are involved. By fairly direct extension, the rules for two indexes apply to three or more indexes, a topic considered in the later sections of this chapter. Also, the early sections refer to numerical matrices, but the same ideas are used to index general arrays of characters, cells, structures, and user-defined objects.

4.1 Sources, intermediates, and targets

A source is any pre-existing array from which we are extracting information, whereas a target is any array (pre-existing or not) into which we are storing information. In the following example, there is a single target `Tg`, and six sources:

```
Tg(K1,K2) = U*S(K3,K4);   % Sources: K1,K2,K3,K4,U,S.
```

However, when MATLAB executes this statement, it also produces a series of intermediate results, e.g:

- `R1,R2,R3,R4` are versions of `K1,K2,K3,K4` suitable for indexing.
- `R5` is the result of `S(K3,K4)`.
- `R6`, the final intermediate, is the product `U*R5`.

The final intermediate `R` may then have to be converted, e.g. reshaped, to fit into the appropriate part of `Tg`. In addition, `Tg` itself may be reshaped by its indexes and by the shape of `R`. Sources themselves are never altered unless they are also targets.

Only sources and targets can be indexed. Implied intermediates cannot be indexed; thus the following example is illegal:

```
Tg(J) = (U*S)(K);
```

This example is a failed attempt to index the matrix product `U*S`. Even if `K` is a valid index of the matrix `U*S`, the process must be done in two statements:

```
X=U*S;   Tg(J)=X(K);
```

4.2 Definitions: puts, gets, and vec

The symbol `->` will mean "puts" or "produces" in the sense that an indexed source `S` puts an intermediate result in `R`. The symbol `<-` will mean "gets" or "becomes" in the sense that an indexed target `Tg` gets the final intermediate `R`.

MATLAB's indexing conventions often require arbitrary arrays to be regarded as vectors. The `vec` operator changes an array to a column vector, by stacking successive columns below the first, which puts the elements in column-major order:

```
    vec([1,2,3; 4,5,6]) -> [1; 4; 2; 5; 3; 6]
```

We will also use "vec" as a verb (forms: vec, vecking, vecked). The `vec` operator is not an explicit MATLAB-supplied function, but it is very easily coded as `M(:)`. For instructional convenience, we can code this operation as a MATLAB function:

```
        function V = vec(M);   V = M(:);
```

For indexing, we usually have no need to vec anything explicitly. MATLAB does it as required, e.g. whenever a matrix or array source `S` is singly-indexed, as in `S(J)` rather than, say, `S(J,K)`, `vec(S)` is used implicitly:

```
                S = [1,2; 3,4];   S(3) -> 2
```

Vecking a matrix involves almost no computation or number-shuffling. MATLAB already stores the entries in column-major order. For example, to change a 30-by-20 matrix to a column 600-vector requires only that the dimensions be changed from 30-by-20 to 600-by-1. The entries of the matrix remain stored in the same order.

4.3 Indexing Sources

The purpose of indexing a source `S`, as in `S(J,K)->R`, is to form an intermediate R from the parts (elements, rows, etc.) of S, with possible permutations and repeats. `S` itself is never altered. Initially, `R` does not exist. It is born when `S` is indexed, and it dies upon further use. This short-lived value of `R` depends only on `S` and its indexes. Unlike indexed targets, indexed sources (and sources in general) must pre-exist. If `S` does not exist, an assignment like `Tg=S(:)` will not produce some default `S` to store in `Tg`; it will produce an error. Likewise, index entries must refer to parts of `S` that exist before the index is used. An index entry that is out of range will not produce some padded value from a temporarily enlarged `S`; again, it will produce an error.

4.4 Indexing Targets

The indexing of targets has many aspects in common with the indexing of sources, but also many differences. The purpose of indexing a target `Tg`, as in `Tg(J,K)=R`, is to add, alter, or delete parts of `Tg`. Indexed `Tg` is usually altered, sometimes only in value, but sometimes in size and shape as well. Therefore, we must distinguish between two shapes of `Tg`: the initial

Tg (just before the assignment) and the resulting or final Tg. At times, initial indexed Tg may be nonexistent, and legally so. At those times, MATLAB creates Tg of sufficient size. An index entry can often refer to a part of Tg that has not been created yet. If this reference creates a larger Tg, any new elements of Tg that have unspecified value are padded with default elements: zeros for numerical data, ASCII zeros for character data (which print as white space), and empty arrays for cell and structure data. The ultimate fate of an indexed target depends not only on the target and its indexes, but also on the intermediate R waiting to be stored in Tg. This intermediate will influence the values in Tg, the size and shape of Tg, and the very legality of the assignment statement. Empty R is used to delete parts of Tg, while non-empty R is used to alter existing elements and augment Tg.

Indexing a target is often the most efficient way to alter it. For example, let v=1:n where n is large. Suppose the nth element of v is to be changed from n to 0. Either of the following two statements will do the job:

```
v=[v(1:n-1),0];      v(n)=0;
```

In the first statement, where v is indexed as a source but not as a target, almost an entire new copy of v is required to form the square-bracket expression. Further inefficiency is incurred when MATLAB creates the new target v, because targets that are not indexed are regarded as new, requiring new allocation of space. By contrast, in the second statement, where the target v is indexed, only the single element v(n) is used, and the old vector v remains in place.

4.5 Single *versus* Double Indexes

A source or target may be singly-indexed, as in S(J), or doubly-indexed, as in S(J,K). For the same S, J in S(J) can have a much different meaning than in S(J,K) or S(K,J). The shape of a single index can infuence the shape of the result, whereas multiple indexes are always used in vecked form. In particular, when a source S is singly indexed, as in S(M), the resulting intermediate has the shape of M unless both S and M are vectors.

4.6 Single Indexes

Single indexes usually refer to elements in vec(S) or vec(Tg):

```
S=[1,2,3; 4,5,6];   S([2,5]) -> [4,3]
```

Note that the expression S([2,5]) contains a single index, namely the vector [2,5]. That vector is a single entity in MATLAB, for indexing as well as other purposes. Therefore, single-index rules apply. The expression S([2,5]), which has one vector index, is not to be confused with the expression S(2,5), which has two scalar indexes. Note also that the result [4,3] has the same shape as the single index [2,5]. Only single indexes can impose this correspondence.

4.7 Double Indexes

The first of a pair of double indexes refers to the rows of `S` or `Tg`. The second of the pair refers to columns.

```
S=[1,3,5; 2,4,6];   S([1,2,1],[1,3]>0) -> [1,5; 2,6; 1,5]
```

Here, the first index vector `[1,2,1]` says "copy rows 1, 2, and 1 of `S` into `R` in that order", whereas the second index vector `[1,3]` says "copy columns 1 and 3 of `S` into `R`, omitting column 2".

4.8 Types of Indexes

Indexes are of five types: empty, logical, ordinal, default, and end.

4.9 Empty Indexes and Intermediates

Empty indexes in sources produce empty intermediates. If multiple indexes are used on a source, and one of them is empty, the dimension of the intermediate may still have have some non-zero dimensions even though the intermediate is empty. For example, if `size(S)` is `[3,4,5]`, a 3-D array, then `S(:,[],:)` will have size `[3,0,5]`, and will be empty. Empty single indexes can be used in targets as follows: `Tg(J)=R` where both `J` and `R` are empty, all of which leaves `Tg` unchanged.

The empty intermediate `[]` is used to delete parts of the target `Tg`:

```
Tg = [ 1,2,3; 4,5,6; 7,8,9 ];   % Tg is 3-by-3.
Tg( :, 2 ) = [];   % Column 2 is gone, Tg is 3 by 2.
```

For more about this feature, see the subsection on default indexes below.

4.10 Logical Indexes

A logical index, or a logical array in general, contains the logical values (logicals) `true`, which is denoted `1`, and `false`, denoted `0`. Logical elements are stored one value to a byte. By convention in this chapter, we will create two logical scalars: `T` for true and `F` for false to distinguish them from the numbers 1 and 0.

Logicals are created by one of the following methods:

- Conversion from numeric data, e.g. `K=logical(A)`, where `A` is a matrix of numbers. `K` will be the same size as `A`, with Fs whereever `A` has 0s, and Ts otherwise.

- Use of the relational operators `( ==,  ~=,  <,  <=, >,  >= )`, e.g. `K = A<B;` where `A` and `B` are arrays of arbitrary numbers and `K <-` a matrix of Ts and Fs.

- Use of logical operators or functions:

```
and(&), or(|), not(~), xor, any, all, exist,
isempty, isinf, isNaN, issparse, isa, etc.
```

- Use of array operations (subscripting, square-bracket construction, and the `cat` function) on existing logical arrays.

In a source index, T means “include”, while logical F means “exclude” or “omit”:

```
S = [1,2,3; 4,5,6; 7,8,9];   J = [T,F,T];   K = [F,T,T];
% Result:   S(J,K) -> [2,3; 8,9].
```

In a target index, T means “put incoming data here”, while F means “leave this part as it is”:

```
Tg=zeros(3,3); Tg([T,F,T],[F,T,T]) = [5,6; 7,8];
% Result:   T <- [0,5,6; 0,0,0; 0,7,8]
```

For either sources or targets, it is the *positions* of the Ts in a logical index that determine which parts of a matrix will be accessed or altered.

In the next section, we will note that for any logical index, there is an equivalent and likely more efficient ordinal index. So why bother with logical indexes at all? The answer lies in the ability to detect conditions concisely. For example, suppose we are given two row n-vectors `x` and `y`. We want to extract values `x(K)` from `x`, but only when those `x(K)` are positive and less than the corresponding `y(K)`.

```
L1 = x>0;       % L1(K)=true when x(K)>0.
L2 = y>x;       % L2(K)=true when x(K)<y(K).
L3 = L1 & L2;   % L3(K)=true when L1 & L2 are true.
z = x(L3);      % z = specified x(K).
```

The entire process can be telescoped into a single statement:

```
z = x( (x>0) & (x<y) );   % Shortcut.
```

These codes illustrate the power inherent in logical indexing.

4.11 Ordinal Indexes

An ordinal index consists of positive integers. For sources, the value of an index entry tells which part (element, row, or column) of the source is to be accessed, while the position of an ordinal entry tells which part of the result those elements will be stored in:

```
S = [11,12,13;    % S([2,1],[1,3]) -> [21,23;
     21,22,23;    %                     11,13]
     31,32,33];
```

For targets, the value of an index entry tells which part of `Tg` is to be altered, while the position of that entry tells which part of `R` will be stored there:

```
Tg = zeros(1,3);   Tg(2:3) = [5,6];   % Tg <- [0,5,6].
```

Ordinal indexes are far more powerful than logical indexes. For any logical index `L` with at least one 1 entry, there is an ordinal index `K` that will do the same job. For example, the second and third statements below are equivalent:

```
S = rand(1,6);   Tg = S([F,F,F,T,T,T]);   Tg = S([4,5,6]);
```

That is, ordinal indexes can cause inclusions and omissions, just as logical indexes can. But beyond the ability of logical indexes, ordinal indexes can specify permutations and repeats, in principle without limit:

```
S = [3,6,9];   % S([1,2,3,2,1]) -> [3,6,9,6,3]
```

Further, when used singly, ordinal indexes can dictate the shape of the result, e.g.:

```
S = [3,6,9];   % S([ 1,2; 3,1 ]) -> [ 3,6; 9,3 ].
```

4.12 Converting Logical to Ordinal

An ordinal index may have no logical equivalent, especially if it implies re-orderings and repeats. But it is always possible to convert a logical index to an equivalent ordinal form. The MATLAB-supplied `find` function will do the job:

```
L = [T,F,F,T,F,F,T];
f = find(L);             % f <- [1,4,7];
f = find(L,1);           % f <- [1], 1st T only.
f = find(L,2,'last');    % f <- [4,7], last 2 Ts only.
```

The logical form is often generated first, because it is the most convenient way to use the results of a logical test on a vector, e.g. `x(v>1)`. However, the ordinal form is often more efficient for subsequent processing, because it may contain far fewer elements.

4.13 Default Indexes and Cross-Section Removal

The default index, denoted ":", acts as the column vector of all integers running from 1 to some contextual maximum value. E.g.:

```
S=[1,3,5; 2,4,6];   S([1,2,1],:) -> [1,3,5; 2,4,6; 1,3,5];
```

In this context, ":" stands for the index vector `[1,2,3]`. Some rules for determining default indexes are:

- A single default index ":" runs from 1 to the number of elements in `S` or `Tg`, i.e. `1:length(vec(S))` or `1:prod(size(S))`.

  ```
  S=[2,4; 6,8];   S(:) -> [2; 6; 4; 8]
  ```

- For doubly-indexed sources, ":" as a first index runs from 1 to the number of rows in the item, while ":" as a second index runs from 1 to the number of columns.

```
S = [  1,2,3;  4,5,6;  7,8,9  ];
S( :, 2 )  ->  [ 2; 5; 8 ];
S( 2:3, : )  ->  [  4,5,6;  7,8,9  ]
```

Default indexes have a special role in deleting parts of arrays. Cross-sections (rows, columns, pages) of an array can be deleted by storing an empty array in them:

```
Tg = rand(10,10,10);    % Tg has 10 rows, columns, and pages.
K = 5:7;                % Index K is [5,6,7].
Tg(:,:,K) = [];         % Pages 5,6,7 are gone; Tg has 7 pages.
```

Note in the last line that the target `Tg` has only one non-colon index, and this is the general rule when storing empty information. The restriction to one non-colon index prevents such errors as `M(2,3)=[]` where `M` is a 3-by-3 matrix. It is uncertain what kind of result one would want from the deletion of a single matrix element, or from any non-cross-sectional subset of elements.

Targets can become empty by removing all cross-sections of a specific type. In the above example, if `K` had been `1:10`, then all of `Tg`'s pages would be deleted, leaving `Tg` empty. However, one property of the default index is that it does not zero out the corresponding index in such a case. That is, the size of `Tg` would be `[10,10,0]`, not `[0,0]` as one might expect. Nonzero dimensions of an empty array are of use when conformality is required, e.g. in matrix multiplication. For example, the operation `A*v`, where A is 3 by 3, requires that `v` have 3 rows. Even if `v` is empty, it must have size `[3,0]` (not `[0,0]`), whereupon the product will likewise have size `[3,0]`. Proper use of default indexes when creating empty arrays will maintain conformality.

4.14 The End Index

MATLAB provides a built-in special scalar index called `end`. In a manner related to default indexes, when the reserved word `end` is used in an index, it denotes the upper limit of the index in question. When `end` is used in an index, it creates an ordinal index. For example:

```
v = [2,4,9,6,8]               M = [ 11,12,13,14; 21,22,23,34 ]
v(2:end-1) -> [4,9,6]             M(end,:) -> [ 21,22,23,24 ]
v(end:-1:1) -> [8,6,9,4,2]        M(:,end-2) -> [ 12; 22 ]
```

4.15 Case in Point: Vectors

Single indexes hold a special place in the MATLAB indexing scheme. MATLAB even provides functions `ind2sub` and `sub2ind` to convert single indexes to and from multiple subscripts. Since vectors are usually singly indexed, they become a special topic. Also, because vectors are the simplest kind of multi-element array, they are used often, taking on an importance worthy of separate discussion.

Vectors with two or more elements can have either row or column orientation. Appending elements by target single indexing preserves the row-column nature of the target:

```
V = [10,20];   V([3;4]) = [30;40];   % V <- [10,20,30,40]
V = [10;20];   V([3,4]) = [30,40];   % V <- [10;20;30;40]
```

Notice that the row-column orientation of the target prevails over the orientations of indexes and intermediates.

Difficulties can arise when the vector is first defined, or temporarily cleared, or when it has been reduced to a scalar or empty by storing empty information in it:

```
clear V;        V(1) = 50;     % V <- 50, a scalar.
V = 10:10:90;   V(2:9) = [];   % V <- 10, a scalar.
V = 10:10:90;   V(1:9) = [];   % V <- [].
```

at which time `V` has neither row nor column orientation. Subsequent extension of `V` by target single indexing will result in a row, regardless of any previous orientation:

```
V=[3;4];   V(2)=[];   V(2)=4;   % V <- [3,4], not [3;4].
```

Also recall that colon notation produces a row, so that initializing `V` as `V=1:9` will produce a row.

While row orientation is easy to initialize and maintain, column orientation is often required by the matrix operations you are using. If a vector `V` is frequently accessed as a column, it may still be stored as a row, and referred to as `V(:)` whenever column orientation is required, but this requires the generation of an intermediate vector with column orientation. Hence, you may prefer to keep `V` as a column, which is easy if `length(V)` is fixed. For varying `length(V)`, which may be 1 or 0 at times (initialization being a frequent case), here are some ways to maintain column orientation:

```
% Code 1: For the case where index K is an arbitrary array.
if isempty(X);           % If X is empty, use a single index to
   V(K) = [];            %    delete the Kth elements of V.
else                     % Otherwise, use vec(X) and double
   V(K,1) = X(:); end    %    indexes on V to ensure column.
% -------------------------------------------------------------
% Code 2: To add to or delete from the end of V,
%            one element at a time.
V = zeros(999,1);   % Preallocate maximum space for V.
n = 0;              % Initialize nominal size of V.
n=n+1; V(n)=x;      % Add an (n+1)st element to V.
n=n-1;              % Delete nth element (n~=0).
% -------------------------------------------------------------
% Code 3: Same purpose as code 2.
V = zeros(0,1);     % Initial V is a empty column.
V = [ V; x ];       % Add an element to column V.
V(end) = [];        % Delete last element from V.
```

Code 1 is versatile, enabling alteration of existing elements, as well as addition and deletion. Its disadvantage is that, should the size of `V` fluctuate widely, much time might be spent looking for workspace to store `V` at its peak sizes. For some applications, it may be best to preallocate space for `V`, as in Code 2. This strategy obliges the user to keep track of the number of elements `n`, but with sufficiently large initial space, `V` never needs overhead to find more space. On the face of it, Code 3 looks simpler than Code 2, but as in Code 1, `V` can grow, demanding more and more space. Further, unlike Codes 1 and 2, whenever an element is added to `V`, a copy of `V` is generated to form `[ V, x ]`, which compounds the overhead problem. Such overhead considerations become serious with intense repetition and/or large magnitude and fluctuation of `length(V)`.

4.16 Thinking in Many Dimensions

An n-dimensional array of numbers is indexed `A(I1,I2,...,In)`, where the jth index `Ij` may be of any type described in the previous chapters. The first two indexes are like the indexes of a matrix, describing the row and column positions of the elements. If you think of each matrix (block of columns) residing on a separate page, then the third index tells which page we are talking about. If you think of each block of pages residing in a separate book, then the fourth index tells which book we are talking about, and so on through shelves, walls, rooms, buildings, cities, counties, states, nations, planets, solar systems, galaxies, universes, and whatever you think lies beyond that.

4.17 Storage and Addressing

No matter how many indexes you use, the storage of the numbers is much the same as for a vector. Recall that a matrix is stored one column vector after another as we proceed across the page. If there is more than one page, the pages will likewise be stored one after the other, and so on. For example, the storage of a 2-by-3-by-4 array is indistinguishable from that of a 24-vector. Vectors are distinguished from 3-D arrays because, off in separate entries somewhere, MATLAB records the dimensions of each array. This uniformity of storage enables the user to address any array as though it were an array of fewer dimensions, just as a matrix can be addressed as a vector. For example, a book can be addressed as a matrix or a vector, so that the various styles of indexing are:

```
% Let i, j, and k scalar indexes.
% Let A be D1-by-D2-by-D3.
z = A(i,j,k);   % z  <-  A row i col j page k.
z = A(i,j);     % z  <-  A row i col j, where A
                %    is treated as a matrix with
                %    D1 rows and D2*D3 columns.
Z = A(i);       % z  <-  A row i, where A is treated
                %    as a vector with D1*D2*D3 rows.
```

Think of indexing a book (3-D) in matrix (2-D) style as tearing all the pages out of the book and taping them side by side, making a single large page or matrix. Likewise indexing a shelf (4-D) like a book (3-D) is like taking all the books on a shelf, ripping off all their covers, and binding them as one huge book. The analogy is not exact because books on a real bookshelf are not obliged to have the same number of pages, nor are the pages obliged to have the same number of rows and columns. But in arrays, strict uniformity is the rule.

4.18 Efficiency

The more dimensions you use for indexing, the more expensive it is to compute the actual location of the desired element in the storage vector. For example, let `Z` be a 4-D array with dimensions `D1, D2, D3,` and `D4`. To address the element `Z(i1,i2,i3,i4)`, where the indexes are scalars, one must locate that element in the `Z` storage vector by computing a single vector index `j` from the four original i's:

```
j = i1 + D1*(i2-1) + D1*D2*(i3-1) + D1*D2*D3*(i4-1)
  = i1 + D1*( i2-1 + D2*( i3-1 + D3*( i4-1 )))
```

This formula can be avoided in certain common cases, e.g. when accessing successive elements of a row or some other cross section, but in the case of random access, indexing effort can easily dominate computation effort if one is cavalier about the use of multidimensions. Using array operations, as opposed to scalar operations in loops, will likely cut down on the burden of dimension. Also, where possible, index the array with as few dimensions as possible. To this end, MATLAB provides a function that computes the vector (single) index of an element of an array of arbitrary dimension and size. The following code does the job for a 4-D array:

```
% Let i1, i2, i3, and i4 be scalar indexes.
sz  = size(A);    % sz is a 4-vector of size.
Ind = sub2ind(sz,i1,i2,i3,i4);
```

If the element `A(i1,i2,i3,i4)` is to be accessed repeatedly, it may now be addressed as the vector element `A(Ind)` without the 4-D burden of indexing.

The function `sub2ind` is also of use when arbitrary parts of an array are to be accessed. As an elementary example, consider the problem of replacing the main diagonal of a large n-by-n matrix `A` with an n-vector `v`:

```
Code 1:   for i=1:n;   A(i,i)=v(i);   end;
Code 2:   A = A - diag(diag(A)) + diag(v);
Code 3:   Ind = sub2ind( [n,n], 1:n, 1:n );
          A(Ind) = v;
```

Code 1 uses a loop and double indexing. Code 2 avoids loops, but at the cost of creating and adding the large intermediate matrices `diag(diag(A))` and `diag(v)`. Code 3 avoids loops, double indexing, and the manipulation of large matrices, while accessing only those parts of `A` that need changing. As a further economy, the first line of code 3 need be executed only once, no matter how many times the diagonal of `A` is changed (provided `A` doesn't change size).

4.19 Dimensionality

MATLAB provides functions that access the dimensions of an array. The expression `size(A)` will return a vector of dimensions, not including trailing singletons. The expression `ndims(A)` will return the number of dimensions of A, again, not counting trailing singletons, i.e. `ndims(A)` is equivalent to `length(size(A))`. However, the number of dimensions is always 2 or greater, even if `A` is vector, scalar, or empty. Here are some examples:

```
Definition of A        size(A)     ndims(A)
 A = [];               [0,0]       2
 A = 5;                [1,1]       2
 A = [4,6];            [1,2]       2
 A = zeros(1,1,4);     [1,1,4]     3
 A = zeros(1,4,1);     [1,4]       2
 clear A;
 A(2,1,1,2,1) = 7;     [2,1,1,2]   4
```

4.20 Scalar Functions of Vectors

MATLAB provides several functions that produce scalar results from vector input. The function `sum` will illustrate how these built-in functions operate in higher dimensions. If `v` is a vector, the expression `sum(v)` produces the sum of the elements of that vector. If `A` is a higher-dimensional array, then `sum(A)` produces the sum(s) along the first non-singleton dimension of `A`, producing an array in which the first non-singleton dimension of `A` is replaced by a singleton. For example:

```
Definition of A     size(A)      size(sum(A))
A=rand(3,5);        [3,5]        [1,5]
A=[1,2,3];          [1,3]        [1,1]
A=rand(1,3,5);      [1,3,5]      [1,1,5]
A=rand(1,1,1,7);    [1,1,1,7]    [1,1]
```

These functions accept an optional second argument (or third argument in the case of max and min) specifying the dimension over which the function is to operate. Thus the expression `sum(A,2)` will produce the row sums of a matrix `A` rather the column sums, even if there is only one column. The two-argument form of these functions gives you absolute control over which dimension is operated upon. The result is an array in which the specified dimension is reduced to a singleton, while the other dimensions are the same as those of the input array. For example:

```
                              A = rand(5,5,5);
M = rand(5,5);                size(sum(A,1)) is [1,5,5]
size(sum(M,1)) is [1,5]       size(sum(A,2)) is [5,1,5]
size(sum(M,2)) is [5,1]       size(sum(A,3)) is [5,5]
```

Some functions that operate in a manner similar to `sum` are listed below.

```
HELP:  all any max min mean median prod std sum
```

4.21 Functions for Multidimensionality

The function `cat` provides a neat way to compose multidimensional arrays. For example, let `M1, M2, M3,` and `M4` be matrices of like size that we wish to join as pages of a 3-D array. The statement `A=cat(3,M1,M2,M3,M4)` does the job. The first argument to `cat` is an integer denoting the dimension along which the joining occurs.

The function `squeeze` squeezes out singleton dimensions, so that, e.g.:

```
A = rand(1,3,1,5);   % size(A) is [1,3,1,5]
B = squeeze(A);      % size(B) is [3,5]
```

Since indexing of multi-dimensional arrays can become quite intricate, MATLAB has an arsenal of built-in functions to help.

```
HELP:  cat reshape permute ipermute size length
       ngrid ndim shiftdim squeeze sub2ind ind2sub
```

4.22 Other Array types

Numeric arrays, which we have been discussing in some detail, comprise only one of several array classes. There are also character string arrays, cell arrays, structure arrays, and object arrays. The syntax of indexing is similar for all of these, although the contents of cell arrays and structure arrays can differ wildly in both size and class. For example, within a cell array, one cell might contain another cell array, while a neighboring cell might contain a single number. Some of the functions that aid in manipulating numerical arrays e.g. `cat` and `squeeze`, are also useful in the the other classes of arrays. In addition, of course, each class of array has its own collection of specialized functions. More detail about character, cell, structure, and object arrays will be given in the next chapters.

4.23 Empty arrays

Yes, you can have multidimensional empty arrays, and in a variety of data classes as well. Regardless of class, a statement of the form:

```
% x has size [ D1, D2, ... Di, ... Dn ]
x( :, :, ... :, 1:Di, :, ... :, : ) = []
% x has size [ D1, D2, ...  0, ... Dn ]
```

renders `x` empty, having zeroed out the ith dimension. However, several things about `x` are not forgotten. For one thing, all dimensions except the one that was zeroed are retained for use in building future arrays by catenation and other dimension-dependent operations. Currently, MATLAB will allow you to catenate an empty array with another array of apparently incompatible dimensions (an empty should fit with anything, right? Wrong!), but it will give a warning message, and presumably in a future MATLAB, such a construct will be illegal. The number of dimensions remembered can be reduced by using fewer default subscripts. In the above example, if `m=n-1`, using `m` indexes instead of `n` produces an empty array `x` of size `[D1,D2,...0,...,Dm*Dn]`.

In addition to past dimensions, the class of the emptied `x` array is also remembered, and in the case of empty structure arrays, the field names are remembered. **Exception:** if a structure array is emptied by removing all of its fields, using the `rmfield` function, the resulting empty array is of class `double`, not `structure`, and its size is `[0,0]`.

Chapter 5

Strings

Along with the floating point number and the integer, the character is a fundamental datum in MATLAB. All told, there are over 30 built-in functions for handling characters, including tests for various types of character data, conversion from character data to numeric and back, execution of strings as MATLAB expressions, and operations on strings, such as concatenation, substring comparison, and pattern-matching. These may be found in the help files under `strfun`.

5.1 ASCII Formats

The ASCII character set is prevalent for input and output in countries that use the roman alphabet. There are two formats for storing ASCII strings. The first format uses one character per byte, which is called *native.* The second format (the default) uses two characters per byte, called unicode. In either format, each character is coded as one of the integers from 0 to 255, which are displayed in character form. To see the ASCII code for a character, convert the character to a numeric type, e.g. `double('A')`, which produces the numeric value 65. Inversely, the integers 0 to 255 can be converted to character, e.g. `char(50)` produces the character `'2'` (as opposed to the number 2). The ASCII character set includes the upper case alphabet (codes 65 to 90), the lower case alphabet (codes 97 to 122), and the numerals 0 to 9 (codes 48 to 57). Also included are punctuation, mathematical operators, printer controls, alphabetic characters with diacritical marks, and various special characters. The ASCII set contains all the printable characters on a standard computer keyboard and more. The unicodes 256 to 65535 are used for non-roman alphabets, and require special settings to be useful. The `char` function will convert all other numeric values (whether those values be huge, fractional, or negative) to 2-byte form with a warning message.

```
HELP: native2unicode unicode2native char
```

5.2 Characters as integers

Character variables and characters *per se* may appear as quantities in mathematical expressions, such as:

```
          c1=char(100);    % 100 is the code for 'd'.
          c2=char(c1+1);   % 101 is the code for 'e'.
```

Characters in mathematical expressions like `c1+1` are converted to the double-precision equivalents of their ASCII codes, and the expression is evaluated in a suitable format. The result of the expression can then be stored in 2-byte format using the `char` function as above. Of course, the results should always be in the integer range 0 to 65535, or with an offset value, you can interpret some of them as negative integers, e.g. -32768 to +32767. So, effectively, a character variable can be treated as `uint16`. However, the use of a character as an index is not allowed.

5.3 Character Arrays

Characters usually occur in rows called strings, which in turn can be in higher dimensional arrays. An explicit string is always bracketed by single quotes:

```
' % between quotes does not indicate a comment.'
```

If the above line appears in a function, its execution causes the line to be printed, just like the result of any expression. The statement:

```
z='zoom';
```

defines the variable `z` as a string of four characters. A matrix of characters is also legal:

```
ArrayType = [ 'real   ' ; 'complex' ; 'string ' ];
```

produces a 3-by-7 matrix of characters, usually thought of as a series of 3 strings. In many ways, manipulation of character arrays resembles the manipulation of numeric arrays. The above example is analogous to the formation of a 3-by-7 numeric matrix using square-bracket notation. The indexing conventions are the same, as is the notation for building larger arrays from smaller ones.

The inconvenience of treating strings as vectors is evident from the above example. In square bracket notation, you must be sure that all strings in the matrix `ArrayType` are of the same length, hence the trailing blanks in two of the strings. The set of strings you need are rarely of equal length without padding. So most strings must be padded when placed in an array, and often de-padded upon use. To avoid the drudgery of padding and de-padding strings, and counting blanks, MATLAB provides several functions that automate the process or make it unnecessary:

```
HELP:  char  cellstr  strcat  strvcat
```

5.4 Strings that Represent Numbers

A string can be converted to a number if it represents one, and a number can be converted to a variety of string formats. For example, the number `pi` can be converted to any of the strings:

```
'3'   '3.14'   '3.14159265'   '  3.14e00'   etc.
```

and strings of the above type can be converted to numbers. The process is simple, but with many options that we will not catalog here. Consult the help files under `strfun` and the references for details.

5.5 The eval Function

A single string can be executed as MATLAB code using the `eval` function. If the string contains a MATLAB expression, `eval` returns the value of that expression. However, the name `eval` is an understatement, because `eval` is not restricted to evaluating expressions. The string argument can contain any long sequence of MATLAB statements separated by semicolons. (MATLAB strings can be very long.) These statements may include `if` statements, loops, function calls, and even commands to the operating system using `!` as the initial character. Additionally, the `eval` function may appear within loops, so it is quite possible to execute entire programs using `eval`.

Be aware, though, that executing strings can affect efficiency. In the following example, four loops were timed using the MATLAB etime function as in the chapter on time and work:

```
% Time for four deliberately slow loops.
t=0:.006:6;  str1='sin';  str2='sin(x)';  str3='feval(str1,x)';
for x=t,  y = sin(x);         end     % 1.15 seconds.
for x=t,  y = feval(str1,x);  end     % 1.64 seconds.
for x=t,  y = eval(str2);     end     % 3.96 seconds.
for x=t,  y = eval(str3);     end     % 5.61 seconds.
```

The degradation of efficiency seen above is really a worst case, because `sin(x)` itself takes hardly any time. When eval and feval (discussed below) are invoked on a costly function, e.g. `inv(x)` for large `x`, their overhead time may hardly be noticed.

5.6 The feval Function

At this writing, MATLAB is being steered away from the use of the `feval` function. Eventually, `feval` will be obsolete, having been replaced by function handles. Since it is still here, and there are many codes that use it, we will describe its use.

If you try to invoke a function called `func` in a code where there is a variable named `func`, the variable will take precedence, and the function will be ignored. The arguments to the function will be treated, mistakenly, as indexes of the variable. When you have a choice of names, it is easy to keep distinct names for your variables and your functions. But when you are writing a function that accepts an unpredictable function name as a string argument, you may not have control over which name the user will put in that string:

```
function Y = F(S,XX);  % S is a string.
Z = XX^2;              % XX is a scalar.
Y = eval([S,'(Z)']);  % Treat S as function.
```

In this example, the argument to `eval` is the string in `S` followed by the string `'(Z)'`. For example, if `S` is `'func'`, then the argument to `eval` becomes `'func(Z)'`, and everything works as expected, because the name `'func'` is not used elsewhere in function `F`. However, if `S` is `'XX'`, which is a variable name in the function `F`, the attempt by `eval` to invoke the function `XX` will instead try to index the variable `XX`. To eliminate this possibility, the `feval` function is provided. Replacing the last line of the above example with:

```
Y = feval(S,Z);
```

forces the string in `S` to be interpreted as a function name. The remaining arguments to `feval` become the arguments to that function. This is one way to treat a string as a function name.

5.7 Function Handles

The preferred way to avoid function-variable conflicts is to make `S` a function handle prior to calling `F`, e.g.: `S=@XX`, making `S` a handle to the function `XX`. Now when `S` is used in function `F`, it will be recognized as a function handle, thus eliminating any confusion with variables. The last line can now be coded as `Y=S(Z)` without the use of `eval` or `feval`.

5.8 Parsing Strings

Interpretation of strings is not limited to MATLAB entities like expressions, statements, and function names. Within MATLAB, you can develop your own language, using MATLAB's string-handling features to help interpret the statements. A useful string function for this purpose is `strtok`. Let's say the elements of your language are 1) variable names of one or more alphanumeric characters, e.g. `pH7`, 2) Numeric constants which always begin with a numeral, 3) operators like + and *, and 4) separators like commas and blanks. The `strtok` function can help to decompose the string into its syntactic tokens. The first line of the code:

```
[ T, R ] = strtok( S, D );
if isletter(T(1)), ... % T is a variable name.
else ...               % T is a numeric constant.
```

will scan the string `S` for any substring delimited by the characters in the delimiter string `D`. The delimited substring, called a token, is returned in the output `T`, and the remainder of `S` is returned in R for further scanning. If, e.g,, you are scanning for a variable name, then the delimiter string `D` might contain all the non-alphanumeric characters in your language. Finally, the `if` statement above will decide if the token is a variable name or a numeric constant.

Another built-in string function, `strrep`, replaces one substring with another:

```
S1 = 'x+x^2+y-x1';
S2 = strrep(S1,'x','z');   % S2 <- 'z+z^2+y-z1'.
```

where every instance of `'x'` in `S1` is replaced by `'z'` in `S2`. This device is especially useful in mathematical deduction and theorem-proving, but it must be used with caution, lest you replace parts of variable names by mistake. In the above example, note that `'x1'` was changed to `'z1'`, which may have been unintended.

The most versatile tools for manipulating patterns in strings are regular expressions, which are found in several modern computer languages. Documentation for the function `regexp` explains how regular expressions are used in MATLAB.

5.9 Treating Files as Strings

The lines of an ASCII data file can be treated as strings, not necessarily of equal length. Each line can be read in turn and processed as follows:

```
fn = 'myfile';          % fn = file name.
fid = fopen(fn,'r');    % Open fn as 'read only'.
while 1;                % Loop 'forever'.
   s = fgetl(fid);      % Read a line into s.
   if ~isstr(s);        % If end of file,
      break;   end;     %   leave the loop.
   % ----- Process string s here. -----------
end;
```

5.10 Strings in Cells

Character arrays contain strings of uniform length. A character array is much like a numeric array, in that every row must have the same number of characters. Shorter strings in the array must be padded with blanks to conform. Often, this requirement is an inconvenience that can be avoided by using cell arrays, which have no restrictions on the sizes of entries:

```
citizen{1,ncit} = 'John Q. Public';
citizen{2,ncit} = '1234 West Main Lane';
citizen{3,ncit} = 'Townsville MD 56789-1234'; ... etc.
```

For some applications, it is convenient to use both character-array and cell-array formats. To facilitate this, MATLAB provides conversion functions:

```
B = cellstr(A);    % Convert character array A to cell
                   % array B, stripping off trailing blanks.
A = char(B);       % Convert cell array B to character
                   % array A, padding with blanks where needed.
```

The help file on strings may be queried using `help strfun`.

Chapter 6

Cells and Structures

Cell and structure arrays can contain any kind of information, including other cell and structure arrays. By nesting cell and structure arrays, one can create any tree-structure type of data organization, where meaningful data (e.g. numeric and string arrays) reside in the leaves of the tree (or in its root tips if you prefer a top-down chart). The programmer is in complete control of how the tree is branched and what kind of information is found in the leaves. This chapter tells how such information is stored and accessed.

6.1 Cells

Cell arrays can contain, in their elements, any kind of data, e.g. numerical arrays, strings, or even other cell arrays. Whenever a datum can contain its own kind in this manner, very careful use of notation is required to specify the level of depth to which one is referring. A cell array is analogous to a tool cabinet with many drawers. You can talk about the whole cabinet, a particular set of drawers, or the contents of that set of drawers. Even though a drawer is full of screw drivers, you cannot drive a screw with the drawer itself. Similarly, even though a cell contains a numerical vector, you cannot do a vector operation the cell itself. You must operate on the vector within the cell. To make the distinction, the cell itself is denoted by its name followed by parenthesized indexes, e.g. `c1(i,j)`, while the content of that cell is denoted with braced indexes, e.g `c1{i,j}`. Braces are also used in explicit definition of cell arrays, much as brackets are used in matrix definitions. The following example illustrates these ideas.

```
c1 = cell(2,3);       % Create a 2-by-3 cell array (empty content).
c2 = c1(1:2,1:2);     % Cell array c2 <- a 2-by-2 sub-array of c1.
c3 = {'a', pi;...     % Define 2-by-2 cell array c3 by a braced list,
      50, 'efg'};     %   which is an explicit cell array.
c3{2,2}(3) = [];      % Remove the 'g' from the string in c3(2,2).
                      %   Note the use of braces and parentheses.
r = rand(99,99);      % r <- square random matrix.
c1(2,2) = r;          % Bad! A cell per se cannot be
                      %   replaced by a matrix.
c1{2,2} = r;          % Good! The content of a cell may be
```

```
                          %   may be replaced by a matrix.
c1{1,4} = c1;             % A complete copy of cell array c1 is placed
                          %   within cell c1(1,4). Because c1 was only
                          %   2-by-3, it has been expanded to 2-by-4.
                          %   c1(2,4) now contains an empty array.
c1{1,4}{2,2}(7,9)         % Display the (7,9) element of the 99-by-99
                          %   matrix contained in the (2,2) element of
                          %   the (unnamed) cell array which in turn
                          %   resides in the (1,4) element of c1.
c1(:,4) = [];             % Delete the 4th column of c1.  This is a
                          %   special syntax for cross-section removal.
c1{1,2} = [];             % Replace the contents of c1(1,2) and
c1{2,2} = [];             %   c1(2,2) with empty.  This does not
                          %   remove the 2nd column of cells from c1.
c1{2,3} = {};             % Replace the contents of c1(2,3) with an
                          %   empty list, i.e. an empty cell array.
```

To carry the tool cabinet analogy to its unnatural extreme, we can imagine that some drawers can actually contain smaller tool cabinets and so on, like nested Russian dolls. Denoting a particular tool from such nested drawers requires a pair of braces for each level of nesting, followed by a parenthesized index specifying the tool. Further, we can imagine a telescoping cabinet design, so that when a row or column of drawers is removed, the cabinet shrinks to conform. Ah, analogies!

A cell array is a (possibly multidimensional) rectangular array, whose size and access to elements are governed by the same rules and prohibitions that govern numerical arrays. However, the distinction between a cell and its content occasionally causes some indexing confusion. Given a cell array `C` of size 2-by-3, storing other dimensioned cell arrays in existing elements of `C` does not change the size of `C` itself:

```
C = cell(2,3);
C{1,3}= cell(1,2);
```

The second statement above does not expand `C` from 2-by-3 to 2-by-4. Rather, it simply places a 1-by-2 cell array into the element `C(1,3)`. The size of `C` remains 2-by-3, unaffected by the size of the content of its elements. To expand `C` to 2-by-4, one might use e.g. any one of the following six statements:

```
C(1,4)=C(1,1);   C(1,4)={[]};   C(:,4)=C(:,1);   % line 1
C{1,4}=C{1,1};   C{1,4}= [] ;                    % line 2
[ C{:,4} ] = deal(C{:,1});                       % line 3
```

In line 1, a cell is defined by another cell, whereas in lines 2 and 3, cells are defined by content. Line 3 is a syntax whereby the contents from one cell array (or sub-array) can be transferred, cell by cell, to another cell array with an equal number number of elements.

6.2 Structures

The structure array, like the cell array, is a high level of data organization, in that its elements can contain a mix of all other kinds of data, and may even contain other structures and cell arrays. A structure contains a set of data items, each item being described by a field name. The item is contained in the corresponding field, just as it could be contained in a cell. For example, the structure `customer` can have fields called `name`, `address`, `telno`, `faxno`, `email`, and `order`. The code for creating a new customer in an existing array of customers might be:

```
customer(n).name = 'O. Leo Leahy'
customer(n).address(1,:) = '123 Mauna Loa Drive';
customer(n).address(2,:) = 'Honolulu HI 99999  ';
customer(n).telno = '(808)555-1234';
customer(n).faxno = '(808)555-2345';
customer(n).email = 'OLLeahy@uhi.edu';
customer(n).order(1).title = 'Webster's II';
customer(n).order(1).ISBN = '0-395-33957-X';
customer(n).order(1).costper = 20.95;
customer(n).order(1).number = 5;
customer(n).order(1).status = 'received';
customer(n).order(1).paydate = '1998Jan02';
```

The fields `name`, `telno`, `faxno`, and `email` contain simple character strings. The field `address` is a 2-dimensional character array that can have as many lines as needed for each customer. The field `order` is itself a structure array with six fields, four of which contain strings, while the other two contain numbers. The next order will be referred to as `customer(n).order(2)`.

The structure itself may be indexed, e.g., when there is more than one customer, as in `customer(217)`. New structures can become elements of `customer` e.g. `customer(n)=newcust;` with the restriction that the incoming structures must have the same field names. Also, the fields can be indexed, e.g., when there is more than one order per customer, as in `customer(9).order(3)`. When a structure is indexed, every element of the structure array must have the same number of fields and the same field names. However, as in cell arrays, there is no requirement to have similar content in any of those fields. For example, if customer 217 is a big spender, `customer(217).order` might be a large structure array, whereas if customer 218 has just subscribed without buying anything yet, `customer(218).order` might be empty.

Often, there is a choice about what should be indexed. Consider an example in least-squares curve-fitting, where there are `n` observations, each with six fields:

```
% obs.indvar  = the independent variable, time.
% obs.depvar  = the dependent variable, concentration.
% obs.weight  = the statistical weight, reciprocal of variance.
% obs.compvar = the computed variable to compare to obs.depvar.
% obs.resid   = the weighted residual given by:
%               sqrt(obs.weight)*(obs.depvar-obs.compvar).
% obs.error   = standard error of obs.compvar.
```

If we expect to process hundreds of observations many times during a fitting procedure, should we index the structure or its fields? Put another way, should we have many `obs` structures, each with six scalar fields, or should we have one `obs` structure with six large array fields? This is an important consideration for efficiency. Numerical software might prefer the fields to be converted to vectors and matrices, i.e., each field stored as a mathematical entity. When the field is indexed, as in `obs.indvar(i)`, etc., each field is stored as an array, which is ideal for processing by numerical methods. As a simple example, the sum of squares of the residuals would be computed as:

```
SSR = sum( obs.resid .^ 2 )
```

By contrast, indexing the structure as in `obs(i).indvar` emphasizes each single observation as an organizational unit, thus breaking up the six arrays, assigning their elements to the structures `obs(i)`. Hence, to compute SSR from an array of structures, we first have to collect the residuals `obs(i).resid` from all `obs(i)` before summing:

```
SSR = sum( [ obs.resid ] .^ 2 )
```

which is a far less efficient process. However, in our customer file (above), it is probably preferable to index the structure, because we want to access all of the information about a given customer as quickly as possible. We are not likely to want the sum of all of our customer telephone numbers, although with some frequency, we might want to sum what they all spent and paid, perhaps calling for a different organization for financial data.

While various parts of a cell array or structure array can be added or removed by indexing, the fields of a structure must be manipulated by name. To add a field, initialize that field in any one of the structure elements. For example, to add a new social security number field to the `customer` array, the statement:

```
customer(6).socsec = '123-45-6789';
```

will put the string `'123-45-6789'` in the `socsec` field of customer 6, and empty matrices in all the other new `socsec` fields. To remove the `socsec` field, use the `rmfield` function:

```
customer = rmfield( customer, 'socsec' );
```

Note: through indexing, a structure can become an empty structure, yet still retain its field names. However, if all fields are removed using `rmfield`, the structure loses its structure status and becomes a generic empty array.

6.3 Converting structures to cells

Structure arrays may be converted to cell arrays using the `struct2cell` function. The syntax is quite restrictive, but also simple. Recall that the structure `customer` described above has six fields: `name`, `address`, `telno`, `faxno`, `email`, and `order`. The fields within the `order` field (`title`, `ISBN`, etc.) do not count as fields of `customer`. They are fields of the substructure `order`. The structure `customer` is converted to a cell array `custcell` by:

```
custcell = struct2cell( customer );
```

If there are 6 fields and 999 customers in the structure, then `custcell` will be a 6-by-999 cell array. In general, the number of fields in the structure array becomes the first dimension of the cell array, and the other dimensions of the structure array become the succeeding dimensions of the cell array. The cell array `custcell` has no memory of the structure name `customer` or its six field names. However, the 999 cells `custcell(6,:)` now contain the 999 substructures from the `order` field, including their field names like `title` and `ISBN`, which do not have to be the same in name, number, or content from customer to customer.

6.4 Converting cells to structures

Cell arrays may be converted to structures using the `cell2struct` function, but this process has more options than `struct2cell`. Suppose we have the 6-by-999 cell array `custcell` (above), from which we want to create the structure array `customer` (above). First, we must choose the dimension of the cell array to be converted to fields. Let's choose the first dimension, namely 6. Then, we need to choose the corresponding 6 field names. (If we had chosen the second dimension,we would have needed 999 field names.) Finally, we invoke the function `cell2struct`. A sample code is:

```
% The 1st dimension of custcell will become fields.
fielddim = 1;
% Define a cell array of 6 field names.
fieldname = {'name','address','telno','faxno','email','order'};
% Convert cell array custcell to structure customer.
customer = cell2struct( custcell, fieldname, fielddim );
```

This code reverses the process of the previous section. In general, the structure will have one less dimension than the cell array from which it was converted. Its size will be the same as that of the cell array except that the dimension `fielddim` will be missing, having been converted to fields.

6.5 help entries

MATLAB provides several functions to aid in the handling of cells and structures:

Cells	Structures	Purpose
`cell`	`struct`	Create
`iscell`	`isstruct`	Test types
`iscellstr`	`isfield`	
`cellfun`		Apply functions
`cell2struct`	`struct2cell`	Convert types
`num2cell`		
`char`		
`cellstr`		
`celldisp`	`fieldnames`	Display forms
`cellplot`		
`deal`	`getfield`	Move contents
	`setfield`	
	`rmfield`	Remove fields

Chapter 7

Objects

7.1 Built-in Classes

Object-oriented programming allows you to speak your own private language, with newly defined classes of data, and newly defined operations to go with them. MATLAB already has several built-in classes. The class of any variable `x` can be determined by the call `class(x)`, which returns one of the strings on the left below:

```
double          -- Double precision floating point array
single          -- Single precision floating point array
logical         -- Logical array
char            -- Character array
cell            -- Cell array
struct          -- Structure array
function_handle -- Function Handle
int8            -- 8-bit signed integer array
uint8           -- 8-bit unsigned integer array
int16           -- 16-bit signed integer array
uint16          -- 16-bit unsigned integer array
int32           -- 32-bit signed integer array
uint32          -- 32-bit unsigned integer array
int64           -- 64-bit signed integer array
uint64          -- 64-bit unsigned integer array
<class_name>    -- Custom object class
<java_class>    -- Java class name for java objects
```

The last two items in the above list are not class names, but rather categories of class names. That is, `class(x)` will never return the string `<class_name>` *per se*, but will return the name of the particular object class to which `x` belongs.

7.2 Object Definition

An object is a structure (as described in the previous chapter) that is a member of a class. Every class has a class name, e.g., `Set`. To define the class of objects called `Set`, make a directory `@Set`, in which the methods (functions) for class `Set` will be stored. Note that the name of the directory (after the `@` symbol) must be the same as the name of the class. Class methods include construction of objects, display of objects, evaluation of expressions involving the objects, and conversion of fields from the class in question to other kinds of data. Some of the field names of an object are determined by the object's class methods. Other field names of an object may be inherited from other classes. Inheritance will be discussed later.

7.3 A Simple Example

MATLAB offers a suite of functions to implement the operations of set theory, namely:

```
intersect ismember setdiff setxor union unique
```

You will see these functions, among others, when you type `help ops` at the MATLAB prompt. The trouble is that all the operations look like generic function calls, having none of the symbolic nicety of, say, numerical algebra. Here, we will use object-oriented programming to "algebratize" the appearance of set operations.

Sample functions for the directory `@Set` are given later in this chapter. In our `Set` example, we have only one field called `set`, so that an object `X` of class `Set` is a structure whose Set-based content resides entirely in the field `X.set`. The content can be anything one chooses. Our choice is either an empty matrix, or a row vector of positive integers, sorted, non-repeating. and not exceeding a given maximum. (This notion is more general than it seems. If you are interested in, say, sets of customers, then structures representing those customers can be put in a structure array, and the integers in our Sets can be their indexes.)

Within the directory `@Set`, there must be a function called `Set` (matching the class name) whose task is to construct new objects of class `Set`. A three-statement example of such a constructor is:

```
function S = Set( X );   % Accept input array X.
S.set = X(:)';           % Store X as row in S.set.
S = class( S, 'Set' );   % Convert S to class Set.
% Note: without this last statement (above),
%    the output S would be an ordinary structure,
%    and could not be processed as an object.
```

However, such a terse constructor could easily produce objects that do not contain rows of sorted, unique integers within the proper range. The function `Set` listed at the end of this chapter insures that the content of the `set` field conforms to specifications, or that an error is raised. In addition, we allow three flavors of definition: 1) by default, 2) from an existing Set, or 3) from any numerical array of positive integers:

```
A = Set;                % A.set <- [ ].
B = Set(A);             % If A is already a Set, either of
B = A;                  %  these copies Set A into Set B.
C = Set([4,1;1,2]);     % C.set <- [1,2,4].
```

The standard operations on objects of class `Set` are:

```
~A (not A): All integers (at or below
      some user-specified maximum)
      not contained in A.
A & B (A and B, set intersection):
      integers in A that are also in B.
A + B (A xor B, set exclusive-or):
      integers in A or B, but not in both.
A | B (A or B, set union):
      integers in A or B or both.
A - B (A minus B, set difference):
      integers in A but not in B.
```

However, the symbols we have chosen for our operations have other meanings in MATLAB. To overwrite those meanings for objects of class `Set`, we must code functions within the directory `@Set` with special function names that correspond to the operators in question, namely:

```
~: not,   &: and,   +: plus,   |: or,   -: minus
```

Codes for these five functions are given at the end of this chapter. For a list of overwritable operations, type `help ops`. Currently, there is no way to override the precedence of these operations, which are retained from their non-overwritten counterparts, e.g. `a&b+c` is interpreted as `a&(b+c)`. Of course, the order of operations may be governed by explicit parentheses in the usual way. Note that you can redefine many operations including "=", fields, and subscripting. If you don't redefine these operations, they retain their usual meanings. For example, by default, you can create a multidimensional array of objects by indexing in the usual way.

Objects are rather private stuff. The fields generated by methods of a class cannot be accessed, except by other methods in the same class. If you want other programs to use the contents of those fields, you must provide methods within the class to convert those contents to accessible form. For example, typing the name of an object with no semicolon will not print the contents of that object, unless you code a function called `display` that allows such output. For another example, if you want to make the vectors in the `set` fields available for general processing, you should write a function called e.g. `double`, that converts an object of class `Set` to an object of class `double` (in this case, a row vector). The last section of this chapter contains sample listings of functions `display` and `double`.

So far, we have discussed the attributes of a single class of objects called `Set`. The sample directory `@Set` includes a constructor (`Set`), a displayer (`display`), a converter (`double`), and four operations (`not, and, xor, or`). However, this kind of single-class application is only beginning of object-oriented programming.

7.4 Precedence of Methods

When more than one class of objects is being processed, these classes and their methods can be related in several ways. Consider the statement `z=f(x,y);` where `x` is of class `X`, `y` is of class `Y`, and functions called `f` appear in both `@X` and `@Y` directories. Which `f` will be used? If no provision is made, then the `f` in `@X` will be used because `x` is the first argument. However, you can insist that, say, `f` in `@Y` be used regardless of the argument order by the appropriate use of the `inferiorto` and `superiorto` functions. See the `help` entries for particulars.

7.5 Inheritance of Attributes

An object belongs to the class that spawned it, and possibly to ancestor (parent, grandparent, etc.) classes. That is, it can contain fields from ancestor classes, and it can be manipulated by ancestor class methods. The class that spawns an object is a subclass of any ancestor classes that pertain.

A child-parent relationship is established in the child's constructor function, by using an extended form of the `class` function:

```
function b = B( f, g, ... p1, p2, ..., pn )
% --- code structure b here ---
b = class( b, 'B', p1, p2, ..., pn );
```

The last statement above converts `b` to class `B` with parent objects `p1` through `pn`. Thus `b` will have its own fields, along with inherited fields from `p1` through `pn`, which, let us say, are from classes `P1` through `Pn` respectively, each with its own directory of methods.

Once the first object of class `B` is established in a session, all other objects of class `B` must have the same attributes *in the same order.* If the function `B` above is first called with parents `p1` through `pn` (in that order) as the final arguments, a subsequent call with the parent objects rearranged is not permitted (parenthood is not commutative), nor can one substitute parents of some new class. Consistency of parent class is strictly enforced.

Another restriction on inheritance: the family tree must indeed be a tree, i.e. no closed loops, i.e. no incest. For example, a grandparent class with two descendant classes, which in turn have a common child class will produce an error message. Likewise, a child with two parents from the same class will not do anything useful. One parent's fields will be overwritten by the other parent.

The fields in object `b` are available only to methods of the class that spawned them. Methods of class `B` can access the fields explicitly defined in function `B` above, but they cannot the access the inherited fields. Likewise, methods of class `P1` cannot access the fields from class `B` (the child class), or from the other parent classes. In addition, the parent objects may have their own parents (grandparents of `b`) whose fields will likewise be passed to `b`. An object can have any number of parents, each of which can have any number of parents, and so on, each layer of ancestors adding to the fields that the object inherits. Again, access to those fields remains restricted to the class from whence they came. A descendant's methods may not operate on an ancestor object, because the descendant's fields are not present in an ancestor object. In contrast, an ancestor's methods may operate on descendant objects, because a descendant inherits all the fields of the ancestor's class, and these fields remain accessible (only) to the ancestor's methods.

7.6 Class Detection

It is often useful to know from which class or classes an object gets its fields. To help, MATLAB provides two functions. The first function, `class`, is used with one argument, namely, the object. It returns a single string containing the name of the spawning class, e.g.:

```
A='word';   B=class(A);   % B <- 'char'.
```

but it will not indicate if there are any ancestor classes. The second function, `isa`, used with two arguments:

```
isa( x, 'B' )
```

is a logical function, returning 1 if `B` is the spawning class *or any ancestor class* of `x`. Note that a seeming contradiction can arise. If object `x` has spawning class `B` and ancestor class `A`, then:

```
isa( x, 'B' )      % This expression is true.
class(x) == 'B'    % This expression is true.
isa( x, 'A' )      % This expression is true.
class(x) == 'A'    % This expression is false.
isa(x,'A') & ~class(x)=='A'   % See below.
```

The `isa` test and the `class` test are interchangeable for a spawning class, but not for an ancestor class. Therefore, the fifth expression above is true if and only if `A` is an ancestor class of `x`, providing a test for ancestorhood.

MATLAB does not provide a direct method for tracing the family tree (classes of fields) of a given object. The function `class` will reveal only the child (spawning) class of an object's family tree, while the function `isa` will confirm an ancestor only if you know the name of the ancestor class in question. If tracing the tree is important, it is up to you to provide a function for each class, so that family trees can be passed from class to class to user.

7.7 Aggregation

In addition to inheritance, MATLAB also supports aggregation, in which an object of one class is contained in a field of an object of another class. Once again, we are faced with a structure-within-structure situation, in which one must be careful to specify the level of embedding one is talking about, although the syntax is essentially that of structures. But beyond the structure syntax, we must also be careful to use the right class of methods on each embedded object because of the above restrictions on field access.

7.8 Private Methods

There is a kind of cloak-and-dagger aspect to all this proprietary use of fields, an aspect that can be applied to methods as well. A class of objects can be created for which no single programmer knows everything about the rules or contents. However, if one chooses, class restrictions can be tossed aside. Class converter functions can be made freely available, so that fields of a class can be processed just about anywhere, even in other class methods. Such cross-talk defeats the purpose of object-oriented programming, which is to encourage the use of only the essential attributes of an object at each level of abstraction.

If, for either security or intellectual purity, you wish to keep some of your methods strictly within the class, you can set up a subdirectory called `private`. Such a subdirectory should not be put in a MATLAB search path. All functions in `private` are available only in `private` itself and the immediate parent directory, but not outside the parent, nor in other subdirectories of the parent. Any class directory, indeed, any directory, can have its own `private` subdirectory. Recall also that a function can be hidden by placing its code in another function, where it is available only within that function.

Within a class directory, functions in the `private` subdirectory have precedence over like-named functions elsewhere, so that the expression `sin(x)` used in a class directory will use the function `sin` from the `private` subdirectory, if a function called `sin` is in there. Outside the class directory, there is no knowledge of that hidden function, so the default `sin` (or some other local precedent) is used.

7.9 What Do You Gain?

As an example of object-oriented programming, consider MATLAB itself. Looking at the outline of built-in MATLAB objects at the beginning of this chapter, we see that all MATLAB objects are arrays. Arrays have certain attributes in common. For example, they can be indexed to get at their parts, and they can be attached to each other by catenation using square-bracket notation or the `cat` function. Treating the other object classes as subclasses of arrays has advantages. The code for the above common attributes can apply to several subclasses instead of recoding with possible discrepancies between classes. If the actual code cannot be reused for all subclasses, say, for reasons of efficiency, at least the concept along with its specifications can remain consistent. This gives system designers and programmers a gold standard to shoot for, namely, consistent behavior between classes. Once that is achieved, the user gains an advantage by having to learn one set of array conventions instead of many. In addition, the programmer can code with customized objects and operations that look like what they do, instead of being restricted to an unsuitable set of built-in conventions. In other words, object orientation encourages a more elegant style of program design, which in turn, eases both software maintenance and the users' learning task.

Because of the privacy conventions, good object-oriented programming style avoids global variables, which are a programmer's bane. Global variables are subject to inadvertent alteration by distantly-related parts of a program, creating bugs that are often hard to trace.

7.10 Sample Directory of Object-Oriented Methods

In this section, we define a directory of the following Set methods, which will reside in the directory called `Set`:

`1) Set.m`	`Define an object of class Set.`
`2) display.m`	`Permit display of contents of Sets.`
`3) double.m`	`Convert a Set to a row vector.`
`4) not.m`	`Define ~ for one object of class Set.`
`5) or.m`	`Define | for two objects of class Set.`
`6) and.m`	`Define & for two objects of class Set.`
`7) plus.m`	`Define + for two objects of class Set.`
`8) minus.m`	`Define - for two objects of class Set.`

The goal of this directory is to provide algebra-like syntax for the following built-in set functions using the indicated infix operators:

```
union(|)  intersect(&)  setxor(+)  setdiff(-)
```

The class name `Set` is spelled with upper case S to avoid conflict with the built-in MATLAB handle-graphics function called `set`.

The global integer `SetMAX` must be initialized by the user before methods in @Set are used. `SetMAX` is the maximum integer permitted in Sets. Without `SetMAX`, the not ( ) operator and the Set-generating function `Set` will not work.

The functions (`double`, `not`, `or`, `and`, `plus`, `minus`) accept and produce single objects only, not arrays, and the function `Set` produces a single object. This does not prevent external use of Set arrays, as in `a(2,2)=Set(1:10);`, `b=[a;a];`, etc. That is, the inputs and outputs of these functions can readily become elements of Set object arrays. Because Set arrays are possible, the function `display.m` has been coded to accept arrays.

It is left as an exercise for the reader to redefine the following logical operators for Sets:

```
<    <=    >    >=    ==    ~=
```

to mean proper subset, subset, proper superset, superset, equivalent, and not equivalent, respectively. (Hint: use the built-in functions ismember and all.) For example, `A<B` should produce logical 1 if all values in A are also in B, and B has at least one other value. Otherwise, `A<B` should produce logical 0.

```
function y = Set( x );
% Set - Define objects of class 'Set'.
% A Set is a structure with a .set field containing
% a row vector of sorted positive integers with no repeats,
% Sets are subject to the usual operations of discrete set algebra.
% The operations defined on Sets are:
%    ~A    (not A)   : integers 1:SetMAX not in A.
%   A & B (A and B)  : integers in A that are also in B.
%   A + B (A xor B)  : integers in either set but not both.
%   A | B (A or B)   : integers in A or B or both.
%   A - B (A diff B) : integers in A that are not in B.
% NOTE: the global positive integer SetMAX is the upper limit
%       of the integer elements in Sets, which must be
%       initialized by the user.   If SetMAX is reduced, previously
%       defined Sets with elements > SetMAX may run into trouble.
% ---------- Input ---------------------------------------------------
% x = an object of class Set, or an empty array, or a
%     numeric array of positive integers, double or sparse.
% ---------- Output --------------------------------------------------
% y.set = sorted x(:)' with repeats removed.
   % ---------- Begin -------------------------------------------------
   global SetMAX;                             % Global max integer
   if nargin == 0 | isempty( x );             % Should y be null?
      y.set = [];                             % Yes.
      y = class( y, 'Set' );                  % Convert y to a Set.
   elseif isa( x, 'Set' );                    % Is input a Set?
      if x.set(size(x.set,1)) > SetMAX;       % Check for legal max.
         error('Set element too large');      % Legal max exceeded.
      end;                                    %
      y = x;                                  % No conversion needed.
   else                                       % Input should be integers.
      if isa( x, 'sparse' );                  % Is x a sparse array?
         x = nonzero( x )';                   % Convert to double.
      elseif isa( x, 'double' );              % Is x a full array?
         x = x(:)';                           % Make a row.
      else error('Non-numeric input to Set'); % Wrong class for x.
      end;                                    %
      x = unique( x );                        % Sort x, unique entries.
      nx = size( x, 2 );                      % How many x?
      if x(1)<1 | x(nx)>SetMAX | ...          % Is any x out of range
         any(logical(rem(x,1)));              %   or non-integer?
         error(['Input to Set' , ...          % Error message.
            'must be +integers <= SetMAX']);  %
      end;                                    %
      y.set = x;                              % Structure y gets x.
      y = class( y, 'Set' );                  % y is a Set.
   end;
```

```
function display(s);
% display - Display an array of objects of class 'Set'.
% This function is called whenever an object expression
% or statement is not terminated with a semicolon,
% resulting in display of the entire array in question.
% display can also be called directly, diplaying only the
% part of the array specified in the call, e.g.:
%       a(3) = b & c   % displays all of a.
%       display(a(3))  % displays only a(3)
   % ---------- Begin -------------------------------------
   nm = inputname(1);   % Get input name of Set.
   if isempty(nm),   nm = 'Output Set';   end;
   str1 = [ '   ', nm, '( ' ];
   str3 = ' ) =';
   d = size(s);         % Vector of dimensions.
   c = ones(size(d));   % Initial counters.
   e = c;               % Permanent row of ones.
   N = prod(d);         % # elements in s.
   for i = 1:N;         % Show each element.
      % --- Show Set name & subscripts ---
      if N == 1;   disp([ '   ', nm, ' =' ]);
      else   disp([ str1, int2str(c), str3 ]);
      end;
      si = prod( size( s(i).set ) );   % Size of contents.
      for k = 1:10:si;   % --- Show contents in rows of 10.
         if si > k+9;   v = s(i).set(k:k+9);
         else v = s(i).set(k:si);
         end;
         disp([ '      ', int2str(v) ]);
      end;
      % --- Update indexes c ----------
      k = find( c<d );
      if isempty( k ),   break;
      else
         k = k(1);        % Index to be increased
         v = 1 : k-1;     % Indexes to be reset.
         c(k) = c(k)+1;   % Increase current index.
         c(v) = e(v);     % Reset previous indexes.
      end;
   end;
```

```
function y = double ( x );               % Appendix 2, page 4
% double - convert objects of class Set to class double.
% ---------- Input -----------------------------------
% x = object of class Set.
%     x.set contains a numeric row vector.
% ---------- Output ----------------------------------
% y = row vector, contents of x.set,
%     now suitable for ordinary numeric processing.
   % ---------- Begin --------------------------------
     y = x.set;   % Convert Set to numeric row.
%
%
function y = not( x );
% not - Prefix 'not' (~) for objects of class 'Set'.
% For an object x of class Set, the expression ~x will produce
% the set containing all integers 1:SetMAX that are not in x.
% ---------- Input ---------------------------------------------
% x = object of class Set.
% SetMAX = global scalar integer, upper limit of the universal
%          set 1:SetMAX.  SetMAX must be initialized by the user.
% NOTE: If x contains elements that are not in 1:SetMAX,
%       an index-out-of-range error will occur.
% ---------- Output --------------------------------------------
% y = object of class 'Set', containing all elements
%     of 1:SetMAX that are not in x.
   % ---------- Begin -----------------------------------------
   global SetMAX;                      % Use the global SetMAX.
   y.set = 1:SetMAX;                   % Initial structure y.
   y.set( x.set ) = [];                % Remove integers in x.
   y = class( y, 'Set' );              % Convert y to Set.
%
%
function z = or( x, y );
% or - Infix 'or' (|) for objects of class 'Set'.
% For objects of class Set, the expression x|y produces
% a Set with all integers in x or y or both,
% also called the set union of x and y.
% ---------- Input --------------------------------------------
% x, y = Sets.
% ---------- Output -------------------------------------------
% z = Set union of x or y.
   % ---------- Begin ----------------------------------------
   z.set = union( x.set, y.set );   % Set union.
   z = class( z, 'Set' );           % Convert z to Set.
```

```
function z = and( x, y );
% and - Infix 'and' (&) for objects of class 'Set'.
% For objects of class Set, the expression x&y produces
% a Set with all integers in x that are also in y,
% also called the set intersection of x and y.
% ---------- Input --------------------------------------
% x, y = Sets.
% ---------- Output -------------------------------------
% z = set intersection of x and y.
   % ---------- Begin -------------------------------------
   z.set = intersect( x.set, y.set );   % Set intersection.
   z = class( z, 'Set' );               % Convert z to Set.
%
%
function z = plus( x, y );
% plus - Infix 'xor' (+) for objects of class 'Set'.
% For objects of class Set, the expression x+y produces
% a Set with all integers in x or y but not both,
% also called the set exclusive-or of x and y.
% ---------- Input ----------------------------------------
% x, y = Sets.
% ---------- Output ---------------------------------------
% z = Set exclusive or of x and y.
   % ---------- Begin --------------------------------------
   z.set = setxor( x.set, y.set );   % Set exclusive or.
   z = class( z, 'Set' );            % Convert z to Set.
%
%
function z = minus( x, y );
% minus - infix 'difference' (-) for objects of class 'Set'.
% For objects of class Set, the expression x-y produces
% a Set of all integers in x that are not in y,
% also called the set difference of x and y.
% ---------- Input -----------------------------------------
% x, y = Sets.
% ---------- Output ----------------------------------------
% z = set difference of x and y.
   % ---------- Begin ---------------------------------------
   z.set = setdiff( x.set, y.set );   % Set difference.
   z = class( z, 'Set' );             % Convert z to Set.
```

Chapter 8

Loops

Beware of loops. Where loops prove necessary, keep their content to a minimum. Take notice of calculations within loops that are completely repetitious, i.e. where the calculation is exactly the same every time through. Move these calculations back prior to the start of the loop where they will be done only once. MATLAB is a partially interpreted language, and some loops must be interpreted anew with every iteration. In contrast, expressions that merely imply loops (e.g. vector and matrix operations) may ultimately be executed by optimized code. In addition, many such expressions involve vector and matrix operations for which your machine may have special hardware to further speed things along. Such hardware will never be used if you insist on looping in an explicit way. The remainder of this chapter, and much of the next, is a case study in alternate looping strategies.

8.1 A Bad Example

```
Example 1:
function y = polyx(x,c)
% y,x are conformal matrices, c is a column vector.
% compute y(i,j) = sum_over_k(c(k)*x(i,j)^(k-1))
[rx,cx] = size(x);   rc = length(c);
y = zeros(rx,cx);
for i=1:rx,
   for j=1:cx,   y(i,j)=c(1);
      for k=2:rc,   y(i,j)=y(i,j)+c(k)*x(i,j)^(k-1);
         end;   end;   end;
```

The function `polyx` in example 1 evaluates a polynomial with coefficients `c(k)` for every element of `x`. The function `polyx` accepts matrix `x` and produces matrix `y`, so the caller can say:

```
y=polyx(x,c);
```

rather than:

```
for i=1:rx, for j=1:cx, y(i,j)=polyx(x(i,j),c); end; end;
```

Functions can accept and produce matrices, which saves the caller a lot of looping. However, this feature is not automatic. You must code your functions in matrix fashion to get any benefit. While the function `polyx` appears matrix-oriented to the user, it contains a triply nested loop (last 4 lines) that renders it exceedingly inefficient. We will reexpress this function with fewer or no loops.

8.2 Attacking the Innermost Loop

Let us start with the innermost loop on `k`. It multiplies each `c(k)` by `x(i,j)^(k-1)` and sums the terms. This is the inner product of the vector `c` with the vector of powers of `x(i,j)`:

```
Example 2: % replaces last 2 lines of example 1.
xp=x(i,j).^(1:k-1);   y(i,j)=xp*c;   end;   end;
```

But we are still generating `y` one element at a time.

8.3 A One-Pass Solution

Let us compute the vector `xp` in one pass for all `x`:

```
Example 3: % replaces the body of example 1.
rc=length(c);          % Polynomial degree + 1.
p=0:rc-1;              % Required exponents of x(i,j).
[rx,cx]=size(x);       % Dimensions of x.
x=x(:);                % Reshape x to a column vector.
x=x(:,ones(rc,1));     % Copies of x into rc columns.
p=p(length(x),:);      % Copies of p into length(x) rows.
xp=x.^p;               % Generate all powers of all x(i,j).
y=xp*c;                % Evaluate all the polynomials.
y=reshape(y,rx,cx);    % Reshape y conformal to input x.
```

This procedure is deliberately spread out so you can see the reason for each step. In practice, the last five lines of example 3 might be replaced by the single statement:

```
Example 4:
y=reshape((x(:,ones(rc,1)).^p(length(x),:))*c,rx,cx);
```

There are times when some degree of looping may be desirable in procedures like example 3. For example, if the dimensions of the above problem are `rc=rx=cx=100`, then the matrices `x`, `p`, and `xp` will require millions of bytes. Time will be used to form these matrices, and to allocate the storage space for them. When such matrices get large enough, MATLAB will spill them onto disk and retrieve them as needed, which requires lots more time.

8.4 Avoiding Large Matrices

Loops can avoid the formation of large matrices, as in example 2 and:

```
Example 5:
rc=length(c); [rx,cx]=size(x);
xp=ones(rx,cx);   y=zeros(rx,cx);
for k=1:rc,   y=y+c(k)*xp;   xp=xp.*x;   end;
```

In example 2, we loop on the dimensions of `x`, whereas in example 5, we loop on the length of `c`. Note that all five methods give the same answer. A good way to choose between seemingly competitive codes is to run timing tests, which we discuss in the next chapter.

8.5 Low-Overhead for-Loops

In recent versions of MATLAB, some effort has been given to reduce the overhead associated with for-loops. To get the benefit of these efforts currently, a for-loop must have the following properties:

1. The only data types in the loop are double, integer (less than 64 bit), character, and logical.
2. Arrays in the loop are at most 3-dimensional.
3. All variables in the loop are defined before the loop begins, and are not redefined within the loop.
4. Variables do not change size (number of elements) or type within the loop.
5. The loop index is scalar, e.g. `for z=1:5`.
6. Only built-in MATLAB functions are called within the loop.
7. `if-then-else` and `switch-case` logical comparisons involve only scalars.
8. Each line of code in the loop contains at most one assignment statement.

Chapter 9

Time and Work

Execution time includes the effort of raw computation, the overhead of processing loops and of forming large intermediate matrices. Sometimes a user is faced with a choice between inefficiencies like looping and forming large matrices. The only way to tell which is best is a time trial. In general, the results such trials will depend on the platform being run.

9.1 Built-In Timers

The function `cputime` returns the number of seconds of cpu (central processing unit) time used since the beginning of the MATLAB session. Time spent on the cpu by a process can be measured as follows:

```
t0 = cputime;      % t0 = cpu time so far.
% Test code goes here.
t1 = cputime-t0;   % cpu time used by test code.
```

Sometimes you want to know how much real time a process takes, i.e. cpu time plus disk access time, and so on. The function `clock` returns the current clock time in a row 6-vector whose elements are year, month, day, hour, minute, and second. The associated function `etime` takes the difference between two clock times in seconds. Clock time spent on a process is measured as follows:

```
t0 = clock;             % Current clock time.
% Test code goes here.
t1 = etime(clock,t0);   % Test code time.
```

The function `tic` resets and starts the “stopwatch” (based on clock time), and the function `toc` returns the clock time in seconds since the last `tic`. The manner in which `tic` and `toc` are used is best illustrated by example:

```
% Compare the execution time of
% statement #1: y=A*B*v, and statement #2: y=A*(B*v)
% where A & B are nXn matrices and v is an n-vector.
tic;                     % Reset and start the stopwatch.
```

```
y=A*B*v;                 % Execute statement #1.
t1=toc;                  % Read the stopwatch.
tic; y=A*(B*v); t2=toc;  % Similarly for statement #2.
disp([t1,t2]);           % Display the times.
```

9.2 Reliability of Timings

Note that elapsed time may be an unreliable measure of your own computing time in time-shared (multi-task, multi-user) environments like workstations and mainframes. However, even `cputime` is not immune to variation. Here are some helpful suggestions about measuring running time. Use your multi-tasking computer at a time when no other jobs are running, say at 2am, when you are alone with the hundreds of other people who are running time trials. Ignore the first timing of newly revised code. The first trial may be contaminated by initialization issues like loading and compiling. After the first run, do several runs on each case to account for time variation.

9.3 Profiling Code

It is often useful to determine which m-files in your code are taking the most time. These statistics might indicate a major inefficiency that can be corrected. Here is a code that will report m-files where most of the time is spent:

```
profile on;
% Execute profiled code here.
profile viewer;
```

The report will list time spent in each m-file and the number of calls between the two `profile` statements. If you suspect that one of your M-files is a bottleneck of computing effort, the `profile` feature can help to find it.

9.4 Allocation Overhead

Whenever you redefine a variable as a non-indexed target, MATLAB must de-allocate the old variable of the same name (if any), find space for the new variable, and set it up. The time for this process is called allocation overhead. It can be reduced by indexing your targets whenever possible. For example, `X=[1,2,3]` incurs allocation overhead, even if the row 3-vector `X` already exists, whereas the statement `X(:)=[4,5,6]` does not. However, even when the target is indexed, if new elements are created, i.e. if `X` is enlarged, allocation may remain a problem. Therefore, if you know that the storage size of a variable will fluctuate frequently, it may be best to pre-allocate space for its largest extent, and use indexes to keep track of its current size.

Chapter 10

M-files and Workspaces

MATLAB code may be stored in M-files, i.e. files with a `.m` suffix. M-files come in two major categories: function files and script files. Functions files are distinguished by an initial line beginning with the keyword `function`.

10.1 Workspaces

10.1.1 The Base Workspace

Your interactive base session has a workspace where your variables reside. If you call scripts from your base session (such calls can be nested, script calling script) those scripts share the base workspace. That is, if a script uses a variable `X`, and the script is called from the base session, it will use `X` from the base workspace, provided `X` is there. Likewise, if the called script creates `X`, it will be placed in the base workspace, where you have access to it.

10.1.2 The Global Workspace

In addition to the base workspace, your master session may have access to the global workspace, which contains all variables that have been declared global in the master session or in a script called by the master session. Once your master session or any of its called scripts issues a `global X` command, you will be using `X` from the global workspace. In addition, any functions that contain a `global X` command will also be using that same `X`. The purpose of the global workspace is to enable the master session, scripts, and functions to share variables without passing those variables as arguments. Here are some common uses for global names:

- Names of files in the MATLAB path are global in the sense that they are available anywhere. Exceptions are M-files that reside in private directories and secondary functions embedded in function files.

- To profile the frequency of M-file use, you can set up global counters named `NRUN` followed by the underline symbol (`_`) and the function name, e.g. `NRUN_func`. The master session would initialize and monitor the values of such counters. The first statements in any such M-file would be the global statement followed by the counter update:

```
function y = func(x);
global NRUN_func (other global stuff goes here);
NRUN_func = NRUN_func + 1;   % Increment counter.
```

- If a huge array must be updated frequently, with only a few entries changing at a time, it is time-consuming to submit the array as an argument to the function that will do the changes. Because the function is changing the array, however slightly, MATLAB must find space to create a complete (and massive) copy of the array within the function. Placing the array in the global workspace avoids the problem.
- If you are running a lengthy iterative process that sometimes aborts or goes on forever, you can place the best results so far in global variables, where you can recover them when the process aborts or when you interrupt it.

Unnecessary use of the global workspace is ill-advised because of the danger of unintended duplicate names. To reduce the probability of such mishaps, global names should tend to be long (at least four letters) and should feature LOTS of upper case letters, in contrast to your local variable names.

10.1.3 Function Workspaces

In addition to master and global workspaces, each function has its own function workspace in which its non-global variables reside. This workspace is created when the function is called, and disappears when the function returns to the caller. The same pattern is observed when functions call other functions, including subfunctions in the same file. The same pattern is also observed when a function is called recursively. Each nested call of a function generates its own workspace.

Functions usually do not share variables of the same name with each other or with the master session unless those variables are global. Almost all non-global communication of variables between functions is in the form of input and output arguments. Exceptions to this form of privacy are described in the next subsection.

10.1.4 Workspace Oddities

Functions can call scripts, in which case the scripts use the calling function's workspace rather than the base workspace. Unlike functions, scripts always share the workspace of the calling process, which is their basic way of receiving and passing on variables in lieu of arguments.

More than one function can reside in a file. Unless you arrange things otherwise, only the first (top) function in the file will be accessible to other workspaces. You can override this privacy by including handles to the secondary functions in the output of the top function. If you code such a file, you should end each function with an `end` statement (not followed by a semicolon) to make your intentions clear. As an example, consider a file with two functions:

```
   FILE #1                    FILE #2
function z1=F1(x1)         function z3=F3(x3)
  y1=x1.^2+x1;               y3=x3.^2+x3;
  z1=F2(x1,y1);              z3=F4(x3);
end;                         function y4=F4(x4)
function z2=F2(x2,y2);         z4=sqrt(x4.^3+y3);
  z2=sqrt(x2.^3+y2);         end
end                        end
```

The two codes produce identical results, but with distinct internal syntax. Note that function `F2` has two input arguments and its code begins after `F1` ends, whereas `F4` has one input and lies entirely within `F3`. Aside from sharing a file, `F1` and `F2` have no access to each others' workspaces. In contrast, function `F4` uses `y3` from the workspace of function `F3`. Since `F4` is nested within `F3`, it shares the outer function's variables whose names are not the same as `F4`'s input arguments.

Deeply nested function calls create a series of workspaces, but only the workspace of the current (most recently called) function is active. The others are dormant. Two of these dormat workspaces are of special importance: the base workspace, and the workspace of the process that called the current function, i.e. the caller workspace. On occasion, we may want a function to have access to these other workspaces. The built-in function `evalin` allows functions to reach into the caller and base workspaces:

```
evalin( 'caller', 'x=6;' );   % Change caller's x.
evalin(  'base',  'y=7;' );   % Change base's y.
```

The built-in function `inputname` allows functions to get the caller's names of the input arguments. This enables a function to inform the caller about certain variables using the caller's terminology. Unfortunately, these two facilities can negate the usual precept that, except for the outputs, a function should not disturb the caller's workspace. The function `evalin` coupled with `inputname`, just like the `global` feature, should be used with considerable caution.

10.2 Self-Documentation

When commenting M-files, it pays to follow protocol. In scripts, the first lines should be comments with `%` as the very first character of each line (no leading blanks, no intervening blank lines, no intervening statements). In functions, these comment lines must immediately follow the function statement. The initial comment lines constitute the help file for this M-file, so that typing `help F` will display the initial comments of the M-file `F`, ending at the first line that does not begin with `%`. The first comment line, called the H1 line, should consist of the M-file name followed by a terse description of what the M-file does. If you enter a directory using the MATLAB `cd` command, and type `help dir`, where `dir` is the directory name, all H1 lines in that directory will be displayed, serving as a table of contents. This is one good reason to group related M-files in separate directories. Alternately, you can put a `Contents.m` file in the directory, which will be displayed instead of the H1 lines. Also, the `lookfor` command scans H1 lines in all directories in the MATLAB path. For example, `lookfor sum` helps to locate M-files that involve sums of some sort.

The initial comments should always contain the following:

1. An initial H1 line.
2. An adequate explanation of what the M-file does.
3. A list of input arguments and their meanings, indicating whether they are essential or optional, and what their default values are. If some input arguments can be omitted, tell what happens if they are. For scripts, these comments should list any variables that have to be predefined in order for the script to work.
4. A list of output arguments, as above. For scripts, these comments should list any variables that are created or altered by the script. Scripts have the nasty habit of leaving the variables they create in the workspace. When coding scripts, make liberal use of the `clear` command to avoid a crowded list of useless variables that might conflict with important stuff.
5. A list of additional functions and scripts required by the M-file.
6. An explanation of *how* the M-file does its job, especially if the methods are good for some cases and not for others.

Beyond the initial comments, a step-by-step commentary is often essential for others to understand your code. Even you will be helped by such comments, when looking back on your own code that hasn't been seen for a while. My rule of thumb is: the right side of the page is reserved for comments, usually one comment per line of code. Code stays on the left, using ellipses and continuations where needed to keep code from impinging on comment space. Occasional detailed comments take entire lines. More comment than code is a good thing.

10.3 Function Arguments

The first line of a function, called the function line, is of the form:

```
function [OutArg1,OutArg2,...]=funcname(InArg1,InArg2,...)
```

where the InArg's (function input arguments) and the OutArg's (function output arguments) can be absent or single arguments or a list of arguments separated by commas. If there are no InArg's, omit the parentheses. If the OutArg list is empty or a single argument, omit the square brackets. If the OutArg list is empty, omit the `=` symbol too.

A function may be invoked by a separate calling statement:

```
[OutArg1,OutArg2,...]=funcname(Inarg1,InArg2,...)
```

where the caller input and output arguments have the same syntax as in the `function` statement above. However, caller arguments do not have to match function arguments in name or number. Order alone tells which caller input corresponds to which function input, and similarly for output. A function that returns at least one argument can also be invoked within a caller's expression, as in `y=2*F(x)+5`. In this context, the caller is understood to be asking for a single output of some sort from function F, in this case a numerical array.

A function communicates mainly through its arguments. Some functions allow an arbitrarily large number of arguments. Other functions allow the caller to omit some arguments, or to use special values like `NaN` or `[]` for specific actions. The function's documentation should make clear when such actions are permitted, and how the function interprets them.

When you code a function that must react to varying numbers of InArgs and OutArgs, you can detect these through two handy built-in functions. The function `nargin` tells how many InArgs the caller has provided, while the function `nargout` tells how many OutArgs the caller has requested.

It is MATLAB's usual policy that a caller's variables should not be subject to change when they are included in an input argument list. If an input argument is not altered by the function, only the argument's address is passed to the function to avoid making needless copies of arguments. In this case, the function uses the original data. On the other hand, if an input argument is altered in the function or the scripts that it invokes, a copy of the argument is used, so that the caller's copy remains unaffected. For exceptions, see the section titled "Workspaces" above.

10.4 Function Handles

As illustrated in the next section, it is often necessary to include function names as input arguments to other functions. There are two ways to do this. The first way is to enter the function name as a string, as in `y=g(x1,x2,'f')`, where `g` is a function of `x1` and `x2` and another function `f`. A better way is to establish a function handle for function `f`, and pass that handle to the function `g`:

```
% A function handle is denoted by the
% name of the function prefixed by @.
fhandle=@f;   y=g(x1,x2,fhandle);
```

Unlike a string argument, a function handle contains enough information to enable efficient access to the function it describes. Function handles are a restricted MATLAB data type. There are no function handle arrays or indexes.

Function handles can provide access to hidden functions, i.e. private functions and subfunctions within a file. To get such access, some normally accessible function (e.g. the top function in a file) with access to the hidden functions must define handles for those hidden functions, and return those handles to the caller. Thus, one can have many handle-accessible functions within a single file.

Function handles are described in more detail in the help files under `function_handles`.

10.5 Variable Numbers of Input Arguments

Sometimes, it is impossible to anticipate how many arguments a caller will submit to a function. As a case in point, consider a root-finder function (call it `Root`) that accepts the name of second function (call it `F`) and a scalar `X`, the aim being to find that value of `X` that will yield `F(X)=0`. `Root` will call `F` repeatedly, varying `X` until `F(X)` is zero or very small. The problem is that functions like `F` often need other values, parameters if you will, as well as the principal variable `X`, and the number of such parameters depends on the nature of the problem. Rather than put these parameters in the global workspace, we would prefer to pass them as arguments to `Root`, which in turn could pass them to `F`, without `Root` caring about how many such parameters there are.

The task is accomplished using the argument `varargin`. Let us say that `F` requires three arguments, the variable `X`, plus the two parameters `Y`, and `Z`, so that `F(X,Y,Z)` is the required call. The call to `Root` might be:

```
 Xfinal = Root( 'F', X, A, B, Y, Z );
```

indicating that `F` is the function to be zeroed, with respect to `X` between limits `A` and `B`. The first four arguments are really all that `Root` cares about. Internally, here is how `RF` treats its arguments:

```
function Xroot = Root( Fh, X, A, B, varargin );
% The while-loop that varies X begins here.
% The following statement calls the function whose
%  handle is Fh with the first
%  three arguments X, A, and B, and the
%  remaining arguments from elements of varargin:
   FX = Fh( X, A, B,  varargin{:} );
% The remainder of the while-loop goes here.
```

In the first line, `varargin` is a cell array, into whose elements have been packed all the extra arguments that the caller provided. In the line of code beginning `FX =`, the expression `varargin{:}` is the list of *contents* of the cell array, i.e., the last part of the caller's input argument list to `Root`, which now becomes the last part of `Root`'s input argument list to `F`. Note that the extra arguments `Y` and `Z` from the caller are never explicit in `Root`'s internal code. Finally, the function line of `F` can be coded as follows:

```
function fx = F( X, Y, Z );
```

oblivious to the fact that `varargin` is acting as a go-between.

If the extra arguments are to be used directly, rather than passed to another function, there are several ways to handle the situation. First, the function statement can include many arguments:

```
function y = F( Arg1, Arg2, Arg3, ...
                Arg4, Arg5, Arg6, ...
                   and so on );
nArgs = nargin;
```

The argument list is continued for as many arguments as the function `F` is ever expected to receive in a single call. In this context, the calling statement can provide fewer arguments than the function statement allows, but not more. The function `nargin` tells `F` how many arguments the caller provided this time, so that `F` can be coded to handle any possibility, except too many caller arguments. Any `Arg`'s that the caller does not provide will be undefined in the function `F`.

Another method for varying numbers of arguments is to use `varargin` as a final argument, to package the end of the caller's argument list in a cell array, as in the function `Root` above. This device places no upper limit on the number of caller arguments. However, if `Root` is to use those arguments directly, it must somehow unpack the cell array, e.g.:

```
Arg1 = varargin{1};   Arg2 = varargin{2};   etc.
```

This device has a danger. It allows the caller to provide more inputs than the function is coded to accept, possibly concealing a bug. The extra elements of `varargin` will simply go unused, with no error message to that effect. For this reason, functions that use `varargin` should check the `nargin` function to see how many inputs there really are.

Another alternative is for the caller to package his arguments in cell arrays or structures, to be unpacked by the called function. One advantage of this approach is that arguments often come in several separate categories. Each category of argument can be packed into a distinct cell array. A possible disadvantage of such packing is that it creates copies of possibly voluminous input data. Unlike varargin, which must be the last item in the function line, prepackaged cell and structure arrays can be entered in any agreeable order, as long as that order corresponds between the call and function statements.

10.6 Variable Numbers of Output Arguments

While the caller can ignore function output arguments by omitting them from the output list in the calling statement, he cannot ask for more outputs than the function provides. However, the function can be coded to provide an arbitrarily large number of outputs using the `varargout` feature (essentially the reverse of the `varargin` feature described above). The cell array `varargout`, if used, must be the final function output argument, and it must be defined within the function, e.g.:

```
function[varargout] = Hcube(x);
% Hcube - produce matrices of hypercube vertices
% of dimensions 1:nargin and sides of length x.
cube = [ 0; x ];                 % Put vertices of 1-D
varargout{1} = cube;             %  cube in 1st cell.
v0 = 0;   vx = x;                % Init. new column.
for j = 2:nargout;               % For jth cell,
   v0 = [v0;v0]; vx=[vx;vx];     %  set up new column,
   cube = [ cube, v0; ...        %  up previous dimension
            cube, vx ];          %  of cube by 1.
   varargout{j} = cube;   end;   % Pack output.
```

If the caller wants the coordinates of, say, all unit hypercubes up to the fifth dimension, the calling statement should read:

```
[c1,c2,c3,c4,c5] = Hcube(1);
```

This example is designed to be simple enough to work easily by hand, so you can see what the code does. In the function line, `varargout` does not require the index `{:}`. MATLAB distributes the contents into the outputs anyway, one cell of `varargin` per output. In the fifth line, the function `nargout` tells how many outputs the caller has specified, just as `nargin` does with inputs. In this application, the jth output contains $(j + 1)2^j$ numbers, so if the caller asks for, say, 20 outputs, there just might be a lack of workspace.

10.7 The Keyboard Feature

The `keyboard` statement stops execution within a function, allowing the session owner to examine and change values that are ordinarily private to that function. In fact, while the `keyboard` statement has control, the session owner can issue any MATLAB statements that are legal at that point. This is a powerful debugging tool, and it can also be used as an input device. When a `keyboard` statement is encountered, it retains control of the session until the user issues a `return` (spelled out). At that time, the function resumes execution from the stopping point. (Authors bias: We would prefer a blank response instead of `return`, especially for keyboard statements that are iterated frequently.) The documentation says "all MATLAB commands are valid" when a `keyboard` statement has control. However, some commands may not do what you expect. For example, when the keyboard statement is in a for-loop:

```
for j=lo:del:hi,   keyboard;   end
```

Changing the values of `lo`, `del`, or `hi` with this `keyboard` statement will not change the sequence of values assumed by `j`. The looping vector `lo:del:hi` is preset before the loop executes. Also, editing the function from within itself, i.e. editing the `.m` file on disk from a keyboard statement within the function, will not affect the current call:

```
function y=multpl(x)
keyboard; y=[x,x];
```

If, from the above `keyboard` statement, you edit the file `multpl.m` so that the last statement reads `y=[x,x,x]`, `y` will still get two copies of `x`, not three. `y` will get three copies of `x` on the next invocation of multpl. Allowing a user to alter the state of a program in mid-run is a tricky business at best. But even with these occasional surprises, the keyboard feature is a gem.

10.8 Debugging

Using `keyboard` statements for debugging is handy when you know where to place them. But too often, bugs arise where they are least expected. For this contingency, MATLAB provides db statements, db being short for debug. The statement dbstop if error, issued in your master session, is especially useful. When an error occurs in a function, this statement puts you in keyboard mode, near the point in the code where the error arose. The effect is the same as if you had placed a keyboard statement in that same spot, with exactly the same privileges.

10.9 Error Messages

Certain errors will terminate the execution of statements, scripts, or functions, and error messages will be issued. These things will happen with no explicit request on your part. Usually, the very statement in which the error occurred is identified, along with the type of error, like zerodivide. You should appreciate that such service has a cost. Somewhere, pieces of code must be tracking, and counting, and checking. Design decisions must be made to insure that such monitoring activity is efficient, and that it doesn't become a prominent part of your computation cost.

Your appreciation for such activity will grow as you write your own files, because then, you will have the option to create your own error messages in addition to those from MATLAB. You will have to decide for yourself how much error-checking to do. Remember: the scariest part of a project is when the error messages stop coming. Be as thorough in your MATLAB self-criticism as it is practical to be. Any less may save you seconds initially, only to cost you weeks when something goes wrong.

As an example, consider a function f(A,B,C) of three real square matrices of equal size. You have decided to check the following:

Are there exactly 3 arguments?
Are they non-empty?
Are they square arrays?
Are they of equal size?

You have decided not to check for complex or non-numerical arrays, because these will cause other error messages as the computation proceeds.

```
function D = f( A, B, C )
if nargin~=3, error('f expects 3 arguments'); end;
if isempty(A), error('empty argument'); end;
vs = [ size(A), size(B), size(C) ];
if length(vs) ~= 6  |  any( vs(1:5) ~= vs(6) ),
   error('f expects square matrices of equal size');
end;
```

Note how all of the desired checks will be done. Although the error message may be slightly less than specific, it will be specific enough for the caller to spot the problem. You could skip, say, the check on equal size, in the conviction that non-conformal matrices will cause errors further into the code. But if the resulting message is likely to confuse the user (especially if the user isn't familiar with your function code), then catching the error at the outset with a user-friendly message is worth while.

If a frequently iterated function has time-consuming error checks, you may be wasting time on largely redundant checking. By the same token, if the checks are removed entirely, you may be wasting time computing nonsense. The question is not whether to check, but where and when. It may pay to move some of the error checks from the function itself to a less busy spot in the code. For example, if the function `g` is being invoked from within nested loops of another function `f`, some of the error checks for `g` might well be placed outside the innermost loops of `f`.

10.10 Warning Messages

In certain cases, MATLAB elects to issue warning messages, which, unlike error messages, do not stop execution. They simply make you aware of suspicious conditions. You too can issue warning messages:

```
dateID = date;   % date is a built-in MATLAB function.
if dateID(4:6)=='Aug',
   warning('User beware! My code is on vacation in August.');
end;
```

Chapter 11

Tricky Stuff

Any language as rich as MATLAB is bound to have its confusing aspects, e.g., code that looks like it does one thing when in fact it does something else. A discussion of some of these confusions will help you to solidify your mastery of MATLAB and to avoid bugs.

11.1 Indexed Targets

The following effects are discussed with numerical arrays in mind, but many of these effects can also occur with non-numerical arrays. Let `T` be a target, and let `J1,J2,J3,...` be ordinal indexes. A statement of the form:

```
T(J1) = X  or  T(J1,J2) = X  or  T(J1,J2,J3,...) = X
```

may have one or more of the following effects:

1. It may assign values to existing elements of `T`.
2. It may convert numerical `T` from real to complex.
3. It may create new `T` of sufficient size.
4. It may expand pre-existing `T` to sufficient size.
5. It may assign new zero or empty elements to `T` by default.
6. It may delete elements of `T` if `X` is empty.
7. It may reshape `X` to fit in `T`.
8. It may regard T as an array of fewer dimensions.

What a wondrous device an indexed target is, that it can accomplish so much. But with every added capability comes added danger. Let us retrace the above list of effects with corresponding errors, any of which may slip by without detection at the time of occurrence, with the results propagating into later calculations:

1. Storing an entry with an incorrect value. For numerical arrays, MATLAB will accept a variety of values including NaN, Inf, and complex numbers.

2. Storing an erroneously complex number, e.g. from `T(i)=sqrt(x)` where `x` should have been positive. One complex entry causes the entire array `T` to be stored as complex.

3. Storing `X` in nonexistent `Tl` (Tee-ell) when it should have been stored in pre-existing `T1` (Tee-one). MATLAB creates `Tl`, and stores `X` there.

4. Addressing `T` with an erroneously large index entry. MATLAB expands the dimensions of `T` to fit.

5. In the same error as above, creating spurious zero entries in the expanded `T`.

6. Storing an erroneously empty matrix `X`. When `X` is stored in a non-empty sub-array of `T`, that sub-array disappears from `T`.

7. Creating a 2-by-2 matrix `X` when it should have been a row 4-vector (a single unintended semicolon can do this) with the intended vector in row-major order. When `T(1:4)=X` is executed, `X` is reshaped to a 4-vector, but now the elements are in the wrong (column-major) order.

8. Accidentally using a fewer subscripts than array `T` actually has. MATLAB treats `T` as a lower-dimensioned array. As a result, the new entries are accepted, but placed in the wrong elements.

Clearly, proper target indexing calls for some caution. There are many ways to err that will not elicit an error message, but will simply proceed erroneously.

11.2 The diag function

Given an m-by-n matrix `X`, where `m>1` and `n>1`, `diag(X)` produces the column vector of main diagonal elements of `X`. Inversely, if `X` is a vector, `diag(X)` produces a diagonal square matrix with `X` on the main diagonal. A conflict arises when the shape of a matrix argument can vary, i.e. when we want to regard `X` as a matrix and extract its diagonal, even when `X` is a vector. To avoid forming an unwanted matrix, a test is needed:

```
if min( size(X) ) == 1,    d = x(1);
else                       d = diag(x);   end
```

11.3 The sum function et al.

Several MATLAB functions, e.g.:

```
any  max  sum   mean    std  cumsum   diff
all  min  prod  median  var  cumprod  gradient
```

work on vector input. In a quite natural way, these functions have been extended to operate on arrays by operating on the first non-singleton dimension, thus producing an another array (possibly a scalar or vector) of results. However, the output can have surprising shape. The following example illustrates, where j is a scalar:

```
R = rand(j,7);   S = sum(R);
```

`R` is a `j`-by-7 matrix, and `S` is a row 7-vector of the column sums of `R`, unless `j=1`, in which case `S` is a scalar sum of the elements of the row vector `R`. To avoid such surprises, a multi-argument form of the above functions, provided by MATLAB, should be favored. The first argument to `sum` may be an array of arbitrary size and dimensionality. Typically, the second argument is the dimension over which you want the function to operate. The output of `sum(A,N)` is an array of at most the same dimensionality as A, but with the dimension indicated by N reduced to 1, e.g.:

```
R = rand(I,J,K);   % R is a 3-dimensional array.
S = sum(R,2);      % S is dimensioned (I,1,K)
T = rand(I,J);     % T is a matrix.
U = sum(T,1);      % U(j) = sum(T(:,j)) even if I=1.
```

This convention has not been extended to sums, etc., over more than one dimension at a time. However, to sum an entire array `A`, use `sum(A(:))`.

The functions listed above do not behave uniformly with respect to their arguments and the size of their output. Consult the help files for particulars.

11.4 Vectors Defined by Colon Notation

For positive `h`, the statement `t = t1 : h : tn;` defines a vector with the `n` successive values `t1`, `t1+h`, `t1+2*h`, out to some final value `t1+(n-1)*h` where `t1+(n-1)*h <= tn` and `t1+n*h > tn` are both true. The final value `t1+(n-1)*h` must lie in the closed interval [t1,tn]. For example, `6:2:9` means the row vector `[6,8]`. So far, we're talking about the literature definition. However, the designers are aware (as we all should be) that the arguments `t1`, `h`, and `tn` are often subject to roundoff error, as are the intermediate elements of the vector. So they have made some compromises that violate the strict interpretation of the rules. When the final value of the vector comes very close to but just beyond `tn` (relative to the range of the vector), it is adjusted to exactly `tn`. If this were not done, some seemingly simple constructs would produce unexpected results. For example, the vector `7/25 : 7/25 : 7` should have 25 elements, but in IEEE floating point arithmetic, `7/25` is slightly too large, so that under the strict cutoff rule, the vector would only get 24 elements. The cutoff rule is relaxed to avoid such unintended truncations. A second kind of rule-bending sometimes occurs when an element of `t` lies very close to zero. Sometimes it will be "rounded" to zero, and sometimes not.

In cases where the arguments have sizable numerical errors, the `linspace` function may be preferable to colon notation, because with `linspace`, you always get exactly the number of elements you expect at exactly the endpoints you specify.

11.5 The Meaning of eps

Floating point systems have a characteric tolerance called machine epsilon, denoted `eps` in MATLAB, with a value near `2.22e-16`. Machine epsilon is the difference between `1` and the next higher number in the system. It is also close to the relative difference between any two floating-point neighbors. The absolute difference between any floating-point magnitude `abs(x)` and its upper neighbor is computed as `eps(x)`.

11.6 Unenclosed Arguments

If the arguments to a function are character strings, and there is no assignment involved (i.e., the function stands alone), then there are two ways to invoke the function. For example, consider the `clear` function:

```
clear('x')    % Enclosed explicit character string.
clear x       % Unenclosed implied character string.
```

Either statement will remove `x` from the workspace. However, the second form of statement will work with any function that accepts string arguments:

```
round(de)          % Proper way to round variable de.
%-----------------------------------------------------------
round de           % Here, de is a character string
                   % with internal codes 100 and 101, making
round('de')        % these last 3 statements equivalent.
round([100,101])   % The result will always be [100,101],
                   % ignoring any variable named de.
```

In books and articles, functions are often written with unenclosed arguments. Avoid that convention in MATLAB. Always enclose the argument when asking for a function value.

11.7 Appending and Removing Vector Elements

In an application where parts of a column vector are being appended and removed in unpredictable fashion (e.g. queuing models), some caution must be taken in those cases when the vector can become scalar or empty:

```
q = [1;2;3;4];  % At opening, 4 customers are waiting.
c=4; n=4;       % c=# customers so far, n=length of queue.
q(1:3) = []; n=n-3;    % 3 customers are served before
n=n+1; c=c+1; q(n)=c;  %    customer #5 shows up.
```

Line 3 of the above code shortens `q` to a scalar. Line 4 makes `q` a vector again, but this time `q`, having no memory of its previous column status, becomes a row. The obvious fix is to code `q` as a row at the start, or if the column status is preferable, change `q(n)` to `q(n,1)` in line 4. A similar problem occurs when `q` becomes empty, i.e., when all customers get served before the next one shows up.

11.8 Variables *versus* .m files

An expression of the form `f`, or `f(i)`, or `f(i,j)`, etc., might be an indexed variable or a function call. If you put such an expression in a script file, assuming erroneously that the supposed variable `f` was initialized elsewhere, you may be invoking a function accidently. Note that a function call does not need an argument list, so even the use of what appears to be a simple scalar like `xyz` may invoke a file of the same name if the scalar `xyz` has not been initialized. Another error of this type occurs if you use the name of a built-in value like `eps`, thinking that it is your variable, but forgetting to initialize it. A similar error occurs when you clear such a variable, then (erroneously) try to use it. For example, the built-in value of `pi` is always there, even after you've cleared your variable called `pi`.

Even script files are not immune to this confusion. One way to print the value of a variable is to type its name alone on a line with no semicolon. However, if you forget to define that variable, and there is a script file with the same name as the supposed variable, odd things can happen. For example, you can get an expected yet spurious printout, along with some unwanted computation, causing several of your variables to change value mysteriously.

Equally strange things can happen if you have defined a variable with the same name as a function or script that you want to use eventually. For example, one frequently-used MATLAB function is `clear`, but this name is not reserved in MATLAB. You can also use `clear` as an array name, but if you do, you will be unable to clear anything in that workspace.

My personal convention for avoiding these embarrassments is to use detailed names for functions and scripts, and short simple names for variables, with plenty of comments to remind me of what they mean.

11.9 eval, feval, and Function Handles

`eval(STR)` treats the string `STR` as MATLAB code, and executes it. This feature appears useful for receiving strings as arguments to your functions, and using them as function names, but eval is not safe in this context:

```
function M = g(funcname,x)
% funcname is a string, x is scalar.
% Compute Lotsa_stuff here.
M = eval([funcname,'(x)']) + Lotsa_stuff;
```

The string `funcname` above is intended to contain the name of a function to be evaluated with argument `x`. But if `funcname` contains the string `'x'` or any other name internal to the file `g`, then funcname will be misinterpreted as a variable name. Thus it might produce a diagnostic (in good cases), or a wrong result that might be hard to detect. This becomes more probable if the function `g` is used as a black box by many people. MATLAB provides two safer constructs for such uses: `feval` and function handles, both of which have been discussed in the chapter on m-files.

11.10 The Global Workspace

Ordinarily, a variable `x` used within a function does not interact with a variable `x` used in the base session or in some other function. Statements like `global x` can alter this convention, making it possible for functions and the base session to share the same variable. Any use or alteration of global `x`, even within functions, refers to that single global value. Clearly, this is dangerous, because functions you didn't write but which you are using as black boxes, may also be using a global version of `x`.

The `global` feature should be used sparingly to avoid unwanted interactions. Functions intended for use by others as black boxes should **LOUDLY** document any global variables they use.

When a variable `x` is global, there are two ways to "clear" it. The statement:

```
clear global x
```

removes `x` entirely. However, if another function is then invoked that contains the statement `global x`, `x` will be reinitialized as an empty matrix. Where there is one `global` statement, there are usually others. (The statement `global x` serves no purpose unless `x` is also declared global somewhere else.) So it is not safe to rely on the non-existence of a global quantity, even if you have cleared it.

In contrast to `clear global x`, the statement `clear x` removes `x` only from the local workspace. Its current value remains in the global workspace, and while it is now unaccessible to the local code, it remains accessible to other functions that declare `x` global. That is, the `clear x` statement causes the local code to forget `x`, and its global status as well. The local code can now create a new local variable `x`, with no effect on the global `x` value.

```
% Example:
global x;     % Global x exists, but is empty.
x=[1,2;3,4]; % Global x is 2-by-2
              % Global & local x are the same.
clear x;      % Local x is now undefined.
              % Global x is still 2-by-2.
x=5;          % Local x is scalar, global x is 2-by-2.
```

In general, it is best to keep local and global variables distinct. Choose names for global variables that you will never use for local variables, e.g. names that end with an underline.

11.11 Long Variable Names

A variable name can be very long, but MATLAB will recognize only the first 63 characters. Two variables with very long names which differ only beyond the 63rd character will be regarded as the same variable. This is not a likely problem, but it is possible if you really try.

11.12 Delimiters in Matrix Definitions

Blanks may be used rather freely within expressions, unless those expressions are within square brackets, as in explicit definitions of matrices. In the latter context, a blank may or may not be an expression delimiter, possibly creating more elements than you might expect. Thus we have the following collection of results:

```
                 Row vector            Elements

           [    x*y+u*v,     x*y-u*v  ]      2
           [   x*y +u*v,    x*y -u*v  ]      4
           [  (x*y +u*v),  (x*y -u*v) ]      2
           [   x*y + u*v,   x*y - u*v ]      2
```

In explicit matrix definitions, MATLAB uses the spaces around `+` or `-` to decide whether that operand is a unary sign as in `x +y` (producing two separate elements `x` and `y`), or a binary operator as in `x+y` or `x + y` (producing a single sum). If you want free use of spaces in an element expression, enclose the element in parentheses, as in the third row vector above.

The end-of-line also behaves in some odd ways. For example, consider the two matrix definitions (listed side by side to save space):

```
DEFINITION OF G          DEFINITION OF H
G = [ [ x1, y1 ];        H = [ [ u1; v1 ],
      [ x2, y2 ];              [ u2; v2 ],
      [ x3, y3 ] ];            [ u3; v3 ] ];
```

The user's intention is to produce a 3-by-2 matrix `G`, using semicolons to stack the rows (which works), and a 2-by-3 matrix `H`, using the commas to place the three column vectors side by side (which fails). Were `H` entered on one line, or were the first two lines ended with ellipses, `H` would indeed be 2-by-3. Alas, neither is done, yet MATLAB accepts the definition of `H` without complaint. However, ends-of-line override the commas in `H`, producing a column 6-vector instead of a 2-by-3 matrix.

11.13 Dots

Dots (periods) are used in the following manner:

1 dot for decimal point,
1 dot for array operations (`.*   ./   .^`),
1 dot for non-conjugate transpose (`.'`),
1 dot for structure field separation, e.g. `f1.f2.f3`
2 dots for parent directory,
3 dots for ellipsis (line continuation).

Despite the numerous uses for dots, conflicts of meaning are actually hard to come by. MATLAB prefers to interpret a dot as other than a decimal point if it can, so that `2.^X` indicates an array of powers, not `2.0` raised to the matrix `X` power. Still, if there is a possible conflict of meaning, use spaces for clarity. For example, let `X` be a matrix:

```
A = 2. ^ X;    % A gets the matrix exponential 2^X.
B = 2.^X       % Each element B(i,j)=2^X(i,j), unclear.
C = 2  .^  X;  % Each element C(i,j)=2^X(i,j), clear.
```

The ellipsis is a peculiar beast. Usually, when coding a multi-line statement, you will get into trouble if you don't use it. But occasionally, you will get into trouble if you do, particularly when assigning sizable matrices explicitly. Rows of a matrix that occupy more than one line of code must be continued with an ellipsis. But even if you end all non-final matrix rows with semicolons, you should begin each new row of the matrix on a new line, *without* an intervening ellipsis. With ellipses, MATLAB may complain about the size of your statement, yet without ellipses, the same statement is processed without fuss. This oddity is encountered when the defining statement is large, where the definition would most likely be stored in a `.m` file.

11.14 The "end" Statement

In several languages, `end` statements terminate the scope of certain control constructs. In MATLAB, these constructs are `for`, `while`, `switch`, `if`, `try`, and `function`. In a long function, an `end` statement may be some distance away from the corresponding control statement. Even with your best efforts at organizing the code, there may be some confusion as to what `end` ends. For local clarity, indentation and comments often help:

```
function z = zap(v);
   z = [];
   for x = v;
      if x > 0;
         y = x;
         while y > 0;
            y = f(y);
            switch y;
               case 1;     y = f1(y);
               case 5;     y = f5(y);
               otherwise; y = -1;
            end % switch
         end % while
         z = [z,y];
      end % if
   end % for
end % function
```

11.15 The “break” Statement

The `break` statement transfers control to the statement following the `end` of the most deeply-nested `for`-loop or `while`-loop that the `break` statement is in. However, `if`, `switch`, and `try` constructs also have `end`’s, which can confuse matters:

```
for i=1:n;
   while L;
      switch E
         Case 'A'
         if L2;
            break;   % A break statement here ...
         end; % if           /
      end; % switch         /
   end; % while            /
   % <--- ... causes transfer to here.
end;  % The for loop will proceed.
```

If a `break` statement occurs in a `.m` file outside of any `for` or `while` loop, it acts as a `return` statement.

The statements `break` and `continue` are sometimes confused. Unlike `break`, `continue` transfers control to the top of the loop and proceeds with the next iteration of the loop, if any.

11.16 Functions and Files

You have coded a function named “mufunc” which begins:

```
function y = mufunc(x)
```

but you have stored it in a file named “myfunc.m”. In MATLAB, you must now refer to this function as “myfunc” at all times. The name “mufunc” in the function statement is ignored. If you invoke “mufunc”, which, say, happens to be the name of a built-in function that you are trying to override, MATLAB will not know about your version, and may proceed with the built-in version without error or warning. File names and function handles are the only means MATLAB has for accessing user-defined functions.

Chapter 12

Desiderata

This chapter describes features that are not currently in MATLAB, but that the author, his students, and his clients have wished for at times. Admittedly, some of these ideas are crude in their present form. Some may prove unworkable or mutually exclusive. Still, if some substantial subset of them were adopted, they would certainly eliminate much of the tricky stuff mentioned in the previous chapter.

12.1 Statements and Comments *versus* Lines of Code

A MATLAB statement should end with a semicolon, and should take as many lines as the author pleases without ellipses. MATLAB should ignore the end-of-line in code. Printing of results should be requested explicitly, e.g. by a statement qualifier like `p:` in `p:a=b*c;`. This is far more civil than the current practice of spewing useless, and often voluminous, output whenever a semicolon is accidentally omitted. Comments should be delimited at both ends, e.g. `%{ Comment %}`, and should be permitted anywhere that a blank is permitted. MATLAB 7 allows this to an extent, but still does not allow code to appear after any part of a comment on the same line. The syntax \%} does not actually end the comment, but rather indicates that this is the comment's last line.

12.2 Functions *versus* Files

Currently, each accessible function resides in a separate file. While there may be several subfunctions in the same file, only the top level function is accessible to the user. If the name of the function and the name of the file disagree, the file name takes precedence.

It would be preferable to divorce the concepts of file and function. A function should be defined as any code that begins with a `function` statement and ends with `end`, whether that code came from a file or directly from the user. Currently, a user can define a function directly using the `inline` function, but only if the function consists of one statement, and has only one output argument.

Like a function, a script should begin with a `script` statement consisting of the word `script` and a name, and it should end with `end`. A file should hold as many top-level functions and scripts as the user cares to put there. In this way, functions and scripts associated with a given process could be grouped in one file without cluttering the file directory with needless entries.

The `help` entry for a file would depend on the number of functions and scripts, and their names. If a file contained exactly one top-level function or script whose name was the same as the file, the `help` command would operate as it does now. Otherwise, The `help` command would present a list of contents with one-line descriptions. More detailed help could be obtained by the command e.g.:

```
help filename:functionname
```

12.3 Functions *versus* Variables

Functions are sometimes confused with variables. Parts of MATLAB help to avoid the confusion, e.g. `feval`, while other parts seem to abet it, e.g. in `f(x)`, is `x` an argument or a subscript? The confusion can be eliminated completely as follows:

1. Currently, array indexes are enclosed in parentheses as in `x(i)`, while cell-content indexes are enclosed in braces as in `c{i}`. In like manner, function arguments should be enclosed in square brackets as in `log[x]`, as multiple output arguments are enclosed now.

2. Function and script invocations should always have brackets, even with no arguments, as in `clear[]`.

3. Special built-in values like `eps`, `i`, `j`, and `pi` should be referred to as functions, e.g. `i[]`. Thus they would no longer interfere with the values or the existence status of user-defined variables with the same names. A user who wants to do it the old way, e.g. use `i` as $\sqrt{-1}$, would simply say `i=i[]` initially.

4. With the preceding reforms, the `feval` feature and function handles could be dropped.

12.4 Functions *versus* Commands

Commands like `clear`, `save`, and `load` accept open arguments, i.e. arguments without parentheses. Confusingly, ordinary functions like `sin` and `log` can do likewise, as illustrated in the previous chapter, usually with odd results. Also, it seems pointless to allow statements like `log d` while disallowing `2*log d` or even `2*(log d)`. Functions should require enclosed arguments. If command/function duality is desirable, then commands should also have enclosed arguments e.g. as `plot` does now.

12.5 Consistency of Built-in Function Names

MATLAB has many functions. Consistency in naming would be an asset. Each suggestion in this section is a small improvement, but as a collective and ongoing effort, the improvements in coding and readability could be considerable.

To conform to the convention for the integer functions `int8`, `int16`, and so on, the floating point functions `float32` and `float64` should replace the current `single` and `double`.

Functions that create data of a certain class should use the name of that class, e.g. `int8`, while functions that convert other data to that class should have a standard prefix like `conv2` (short for "convert to") as in `conv2int8`. A function so named would convert to `int8` any data that could be sensibly converted. If it is desired to have a function that converts from one particular class to another, say `char` to `cell` or the reverse, then the function name should consist of the source class, the numeral 2, and the target class, as in `char2cell` or `cell2char`.

12.6 The "rigid" Attribute

Indexing a target (e.g.) can produce a multitude of unwanted results: wrong sizes, data types, shapes, values, and dimensionality. There should be the option to hold specified properties of an array unchanged. The pair of statements:

```
rigid A;
unrigid A;
```

would control the process. For example, the statement:

```
rigid A   size [3,5]   type 'real';
```

would guarantee that the array `A` remain a real array of fixed size until an `unrigid A` statement occurs, or until the scope of the process ends (e.g. an exit from the function where `rigid` occured). While `A` is rigid, any attempt to change the size of `A`, e.g. to overwrite `A` with nonconformal data as in `A(:,:)=rand(4,9)`, or to store complex numbers in real `A` would raise an error. In MATLAB 7, logical variables are treated this way to an extent. Any attempt to store a non-logical element in a logical array causes an error. In contrast, storing a complex element in a real array is permitted, converting the entire array to complex. The user should have the option to prevent such events.

12.7 Special Values

There should be at least one special numerical value which, when encountered in an expression, stops the proceedings and returns to the user with an error message. This value, denoted `WronG`, may be explicitly stored or detected by the use of special syntax, but any other use would cause an error. The purpose of, say, `x(5:7)=WronG` is to guard against further use of `x(5:7)` until those elements have been properly initialized. Likewise, the purpose of the function `isWronG` is to detect the presence of `WronG`, to help insure proper initialization. The logical function `isWronG` accepts an array argument X, and produces an array with 1's where `WronG` appears in X. Thus, the expression `any(isWronG(X(:)))` is 1 if any value in array X is `WronG`. `WronG` should be distinct from `NaN` and `Inf`, values which would allow the computation to proceed at great expense and no benefit whatever. Also, NaN and Inf can be produced by legitimate computation, whereas WronG, when met in an expression, would be an absolute guarantee of some initialization problem.

In addition to the value WronG, there should be logical control values:

```
NaNArith  InfArith  ComplexArith
```

providing the option to disallow arithmetic in the base session or in any user-written function for which NaN's, Inf's, and/or complex quantities are not legitimate. For example, ComplexArith=0 disallows computations with complex quantities within a given workspace, precluding the likes of sqrt and log with indadvertently negative arguments. At present, computations with NaN and Inf can be trapped, but only in debugging mode.

Currently, the functions max and min treat the value NaN as a missing value to be ignored. However, NaN can also be produced by erroneous computation like 0/0, which will then be interpreted wrongly as a missing value, thus concealing a bug. There should be a special value called Missing, reserved for missing values in statistical operations. Like WronG but unlike NaN, Missing could be specified only by direct and deliberate definition, not by errant arithmetic. The logical function isMissing would be used to detect the presence of the value Missing.

12.8 Indexed Expressions

At present, while it is legal to index an explicit array as in A(I,J), it is not legal to index a more extended expression as in:

```
Z = (X\Y)(I,J);
% which is equivalent to the three statements:
W = X\Y;   Z = W(I,J);   clear W;
```

The syntax in the first line above should be legal. Indexes following a right parenthesis in this manner have no conflicting meaning. Granted, there are times when, with sufficient cleverness, you can generate only the relevant part of a matrix. But there are many examples, like that above, where generating a larger matrix, extracting the desired part, and discarding the rest is the most efficient thing to do.

12.9 Embedded Assignments

It is illegal to store intermediate results "on the fly", e.g., as in:

```
Z = A * ( Y = B+C );   % which is equivalent to ...
                       %    Y = B+C;   Z = A*Y;
```

The value produced by the embedded assignment (Y=B+C) is simply the value of Y. The embedded assignment creates no conflict with other syntax. MATLAB already differentiates the assignment symbol "=" from the logical equality symbol "==". Thus, the embedded assignment Y=B+C cannot be mistaken by MATLAB for the logical comparison Y==B+C.

When used together, embedded assignments and indexed intermediates can produce more concerted and more efficient code:

```
z = ( (X=(A\B)(I,J)) * (Y=(C\D)(I,J)) )(1,1);
% would replace the seven statements:
R = A\B;   X = R(I,J);
R = C\D;   Y = R(I,J);
R = X*Y;   z = R(1,1);   clear R;
```

12.10 Embedded Conditionals

The word "`if`" appears exclusively at the beginning of an `if` statement. In other languages, it has also been used to return a value, much as a function would do, e.g.:

```
Z = Z+W + (U+V)*(if q, A; else B)*(X+Y);
% would replace the if statement:
if q, Z = Z+W + (U+V)*A*(X+Y);
else  Z = Z+W + (U+V)*B*(X+Y); end;
```

The embedded `if` can eliminate the choice between coding duplicate expressions or creating useless intermediate variables.

12.11 Output selector function

There should be a function to select the nth output argument of a function that produces multiple outputs, enabling us to use such a function in an expression no matter which output we use. For example, if we want to use the mth of n outputs from the function `svd` in an expression without storing it explicitly:

```
B = A * argout(svd(X),m,n);
```

The argument `n` above would be the number of output arguments you want the function `svd` to produce, since the meaning of an output can depend on how many outputs are requested.

12.12 Bits

There are multidimensional arrays of numbers, characters, etc. In similar fashion, there should be arrays of bits. That is, the status of bits should be on a par with the status of numbers, strings, and other data classes. The current convention of manipulating bits within integers makes the syntax for bits considerably messier than the syntax for numbers and characters.

12.13 The find Function

At present, the `find` function is less than optimal for n-dimensional arrays where $n > 2$. Dimensions beyond the second are simply collapsed into the second, so that e.g. a 2-by-3-by-4 array is regarded as a 2-by-12 matrix, leaving the user to iron out the indexing. The extension of `find` to higher dimensions is obvious, and should be implemented.

12.14 The linspace Function

The function `linspace` is simplicity itself. The expression:

```
linspace(a,b,n)
```

produces a linearly-spaced row vector running from exact scalar `a` to exact scalar `b` with exactly `n` elements. One exception is `n=1` when $a \neq b$, which makes it impossible to satisfy all of the above conditions. (Currently, the result of this case is `b`, but my own preference is to produce an error for `n=1`, unless `a==b` is true.) The other exception is `n=0`, which is interpreted as `n=1` when it should really produce an empty result. Also, the linspace routine should be extended in the obvious way to handle conformal array arguments `a` and `b`, producing an array requiring one more index:

```
v1=[1,2];   v2=[3,6];  M=linspace(v1,v2,3);
Result:  M = [ 1,2;  2,4;  3,6 ]
```

12.15 The logspace Function

The goal of the function `logspace` is, presumably, to provide a linspace-like service for log spacing. However, the function `logspace` now requires as input the *log10* of the first and last elements of the output, thus introducing the possibility of roundoff error in the end values. Further, and most bizarre, if `b=pi`, it must be entered directly, without taking its log10, thus introducing a discontinuous requirement where there need not be any. Consequently, without prior knowledge about `b`, each new `b` must be tested for safety somewhat as follows:

```
if b==pi;  y=logspace(log10(a),b,n);
else;      y=logspace(log10(a),log10(b),n);   end;
```

The `logspace` function should behave as much like `linspace` as possible, i.e. the arguments should be first value, last value, and number of values, e.g.:

```
v = logspace(1,64,7);   % v <- [1,2,4,8,16,32,64].
```

There should be no roundoff at the end points, and no caller concerns about `log10` or `pi`. Of course, for log spacing, the two limits should have the same sign.

12.16 Keywords

There are several keywords in MATLAB that may not be used as variable names. The keywords can be listed by the function `iskeyword` with no arguments. While there is a need for keywords to indicate certain syntax structures like the `if` construct, there is no pressing need for any keyword to be reserved. Minor changes in MATLAB syntax could eliminate the reserved word entirely, making it clear by syntax alone whether a name is a keyword or a variable or a function.

12.17 The "local" Statement

One of the awkward aspects of writing scripts is the danger of conflicting with variables elsewhere in the workspace. When writing a script, one may not know much about the invoking workspace, and in fact there may be several such workspaces in the course of a MATLAB session. Another awkwardness is the creation of variables that are useful only to the script. The script must either clear these variables upon return, or leave them around to clutter the workspace.

A cure for both ills is a block of code delimited by the keywords `local` and `end`. The `local` statement lists all variables local to that block, variables which will not be confused with those outside the block, and which will be cleared upon exit from the block. Any variables used in the block that are not in the `local` statement are treated as belonging to the workspace at large, as all script variables are now. In the example below, the author of the script need not be concerned that the caller has has variables called `x` and `r`. The `local` statement can impart to a script some of the safety and privacy of a function without lengthy argument lists.

```
% Example: this is the start of a script file.
% u and y are from the caller's workspace.
% param & func are defined functions.
local x r;            % Local x & r are not known to the
   x = param(u,y);    % caller's workspace, and will not
   r = y-func(x);     % disturb caller's x & r if any.
   s = sum(r(:).^2);  % New s is known to the caller.
end                  % Local x & r are now cleared.
```

12.18 Global Labels

The global statement can be made much safer than it is now. Every global statement should refer to a specific region identified by a label, e.g.:

```
global 'G1'  u v;
global 'G2'  x y z;
```

so that global variables `(u,v)` would be shared only with functions that used those variables under the global label `'G1'`, and similarly for the variables `(x,y,z)` under global label `'G2'`. Labeling global variables vastly reduces the probability that those variables would be used by other functions in a conflicting manner. A statement called traceglobal should list all workspaces that currently recognize a given global label, so that use of the label by unintended functions could be detected.

12.19 Initialized Persistent Variables

Persistent variables are currently initialized as empty. A more useful convention is to allow the code to specify the initial value, which would follow the variable name in the `persistent` statement:

```
        persistent smallprime [ 2 3 5 7 11 13 17 19 23 29 ];
```

12.20 More "end" Statements

One way to clarify code, and to enable MATLAB to give more precise diagnostics, is to replace the `end` statement by distinct statements, e.g.:

```
endfor     endwhile  endif
endswitch  endtry    endfunction
```

which would end the various control syntaxes in the obvious way. An alternative to renaming the `end` statement is given in the last section of this chapter.

12.21 The "rename" Command

It is sometimes desirable to change the name of an array. An array may have an unwieldy name, particularly if it has been loaded from a file whose name is some kind of extended mnemonic. Another case where name change is desirable is a utility script file that alters an array, and does so without forcing other copies of the array to be generated, as a function call would do. Rather than change the script code for each input array, one would prefer to change the name of the array to conform to the script. Currently, changing the name of an array involves creating a copy of it, as in:

```
A = Unwieldy_Name;   clear Unwieldy_Name;
```

which can take considerable time if the array is large. Instead, the command

```
rename Unwieldy_Name A;
```

would alter only the symbol tables, leaving the array undisturbed,

12.22 Getting Out of keyboard Mode

There are various situations in which you can find yourself in keyboard mode, e.g.:

1. when a `keyboard` statement is executed.
2. when you issue a `dbstop if error` command, then an error occurs.
3. when you issue a `dbstop if warning` command, then a warning occurs.

As advertised, you can then issue any locally-valid MATLAB statement, with one notable exception. The `return` statement, instead of having its usual meaning: "exit from the function", now means "proceed with execution from the current point in the code". If the current point in the code is in a lengthy and erroneous loop, you may wish to return from the function in the usual sense, but there is no way to do that, because the `return` statement no longer has that meaning.

The `return` statement should mean "exit from the function" even in keyboard mode. The command to proceed from the current statement should be changed from `return` to `proceed`.

12.23 "goto" and Statement Labels

This section describes a semi-desideratum, one that several of my students (especially FORTRAN users) have inquired about. MATLAB, thankfully, does not allow statements of the form `goto L`, where `L` is a statement label, and where the statement labeled `L` can be anywhere in the code. Such a construct would enable us to write the most involuted, indecipherable, and uncheckable kind of code (not that WE would, of course). However, it is possible to have a tamed version of `goto` that does not permit the legendary abuses, by imposing the following conditions.

1. A `goto` statement can only go to a `from` statement, which must lie further down in the code.
2. All `goto` and `from` statements must be labeled.
3. A `from` statement must list all `goto` statement labels that can transfer to it, thus clarifying how the statement can be reached. That is, a `from` statement can only be reached from the statement immediately above it in the code, or from its explicit list of `goto` statements.

An example is given below.

While we are introducing statement labels for `goto`, we might consider requiring them in other cases. One cure for the ubiquitous and often confusing `end` statement is to insist that all statements requiring an `end` be labeled, and that the `end` tell which statement it is ending. Here is an example of the above ideas:

```
for1: for k=1:n;
   if1:  if x;
   gt1:     goto from1;   end if1;
         ... % More code here.
   if2:  if y;
   gt2:     goto from1;   end if2;
     ... % More code here.
end for1;
from1:  from gt1 gt2;
```

This relatively civilized `goto` could either obsolete the `break` and `continue` statements or simply provide lots more versatility. In particular, breaking out of deeply-nested loops can look a lot neater with `goto`. The labeling conventions above could make code more readable, and could help in providing clearer diagnostics about syntax, like which specific statement is missing an `end`, or which `end` is missing an antecedent.