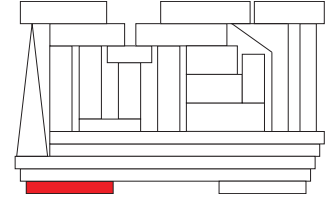


Appendix A



Matlab & Octave: a brief introduction

Matlab (a **portmanteau word**¹ formed from **matrix** **laboratory**) is a powerful high-level programming language marketed by The MathWorks, and is available at <http://www.mathworks.com>. Though expensive², Matlab has become (for small-scale problems³) a de facto industry standard for, among other things, linear algebra, data analysis & visualization, control design, system identification, and optimization.

GNU Octave is a powerful, free, user-developed alternative to Matlab that is mostly compatible with Matlab syntax, and is available at <http://www.octave.org>. A distinct advantage of designing all codes you write in Matlab syntax to run in both Matlab & Octave is that both you and others are assured free access to a legal copy of Octave for any computer platform that you might someday use.

All of the numerical algorithms presented in Matlab syntax in the present text are designed to run in both Matlab & Octave (with a little work, yours may be developed in a similar manner). Thus, you have the choice when following this text to support either high-end, industry-standard commercial software or free, user-developed open-source software, per your individual preference. Please contact the author if you encounter coding errors running any of the algorithms presented in this text in recent versions of either language.

Many numerical problems encountered in science, engineering, and other disciplines are indeed fairly small, and may thus be addressed well with Matlab & Octave. Both languages are also quite useful as intuitive programming languages in which one can experiment, on small-scale systems, with maximally efficient numerical algorithms designed for large-scale systems in an interactive, user-friendly environment in which plotting the simulation results is especially simple. It is thus highly recommended that those who develop and use numerical algorithms today acquire and use (when appropriate) either Matlab or Octave. However, large-scale systems which require intensive computations are much more efficiently solved in compiler-based low-level languages, such as Fortran and C, in which the programmer has much more precise control over both the memory usage and the parallelization of the numerical algorithm. Conversion of numerical algorithms from Matlab/Octave syntax to Fortran or C syntax is straightforward (see §11).

Note that, if you have a fast connection to the internet and access to a unix-based **server** with Matlab already installed, it is possible to run Matlab remotely on the server from your own computer (referred to in this setting as the **client**), using a standard communication protocol known as **The X Window System**

¹ A portmanteau word is formed out of parts of other words, which is common in the naming of computer hardware and software. For example, **Fortran** is a portmanteau word formed from **formula** **translation**, **codec** from **coder/decoder**, **voxel** from **volumetric pixel**, etc. Such words are often formed informally as new technology is developed, then become established through usage.

² Note that the student edition of Matlab is available at many university bookshops at a significant discount.

³ A useful definition of a **small-scale numerical problem** is one that takes a significantly longer time to code than it does to run.

(a.k.a. **XWindows**) to transmit the graphical output of Matlab from the server to the client. If you choose this option, it is recommended that you start Matlab on the server in a unix window with the command

```
matlab -nojvm -nosplash
```

in order to run Matlab directly in the unix window without its communication-intensive **graphical user interface (GUI)**. Users of the unix shells⁴ **csh** or **tcsh** can set this to the default action when the command **matlab** is typed by putting the following command in their `~/.cshrc` file on the server:

```
alias matlab 'matlab -nojvm -nosplash'
```

Users of the **sh**, **ksh**, or **bash** can set this by putting the following in `~/.shrc`, `~/.kshrc`, or `~/.bashrc`:

```
alias matlab='matlab -nojvm -nosplash'
```

Note that, to get XWindows to successfully port the graphics from a server with an IP number 987.654.32.10 to a client with an IP number 123.456.78.90 (assuming the **csh** or **tcsh** is being run on both), you might need to run the following command on the client

```
xhost +987.654.32.10
```

and the following command on the server (before starting Matlab)

```
setenv DISPLAY 123.456.78.90:0.0 (or :1.0, depending on the configuration of the client).
```

Fundamentals of both Matlab and Octave

Installing either Matlab or Octave is straightforward, following the instructions distributed with each package. Once you get Matlab or Octave running, take it for a test drive, and you will most likely find that no manual or classroom instruction on either language is necessary. Most of the basic constructs available in these languages (primarily for loops, if statements, and function calls) can be understood easily by examining the sample codes available in this text. Also, the (lowercase⁵) built-in command names in Matlab/Octave are all intuitive (**sin** for computing the sine, **eig** for computing eigenvalues/eigenvectors, etc.), and extensive online help for all commands is available in both Matlab and Octave by typing `help <command name>`, with even more information on both languages available on the web. These help pages also point you to several related commands, which can be used to learn what you need to know about any given aspect of either language very quickly. To help get you off to a fast start, we now introduce some of the fundamental constructs used by both Matlab and Octave, then explain some of their more subtle features.

To begin, Matlab or Octave can function as an ordinary calculator. At the prompt, try typing

```
>> 1+1
```

Matlab or Octave should reassure you that the universe is still in good order. To enter a matrix, type

```
>> A = [1 2 3; 4 5 6; 7 8 0]
```

Matlab/Octave responds with

```
A =  
    1     2     3  
    4     5     6  
    7     8     0
```

By default, Matlab/Octave operates in an **echo mode**; that is, the elements of a matrix or vector will be printed on the screen as it is created. This behavior quickly become tedious—to suppress it, type a semicolon after the command; this also allows several commands to be included on a single line:

⁴In order to customize your unix environment, you first need to know what **shell** (the protocol defining the set of commands available at the unix command line) you are running in (type `echo $SHELL` to find out).

⁵The names of the codes developed in this text are mixed case, to distinguish them from the (lowercase) built-in commands.

```
>> A = [1 2 3; 4 5 6; 7 8 0]; x = 5;
```

To put multiple commands on a single line without suppressing echo mode, separate the commands by commas. Matrix elements are separated by spaces or commas, and a semicolon indicates the end of a row. Three periods in a row means that the present command is continued on the following line, as in:

```
>> A = [1 2 3;      ...
        4 5 6;      ...
        7 8 0];
```

When working with long expressions, the legibility of your code can often be improved significantly by aligning entries in a natural fashion, as illustrated above. Elements of a matrix can also be arithmetic expressions, such as 3π , etc.

Matlab/Octave syntax has control flow statements, such as `for` loops, similar to most other programming languages. Note that each `for` must be matched by an `end`. To illustrate, the commands

```
>> for j=1:10; a(j) =j^2; end; b=[0:2:10];
```

build row vectors (type `a,b` to see the result), whereas the commands

```
>> for j=1:10; c(j,1)=j^2; end; d=[0:2:10]';
```

build column vectors. In most cases, you want the latter, not the former. *The most common mistake made in Matlab/Octave syntax is to build a row vector when you intend to build a column vector*, as they are often not interchangeable; thus, pay especially close attention to this issue if your code is misbehaving.

An `if` statement may be used as follows:

```
>> n = 7;
>> if n > 0, sgn = 1, elseif n < 0, sgn = -1, else sgn = 0, end
```

The format of a `while` statement is similar to that of `for`, but exits at the control of a logical condition:

```
>> m = 0;
>> while m < 7, m = m+2; end, m
```

A column vector y can be premultiplied by a matrix A and the result stored in a column vector z with

```
>> z = A*y
```

Multiplication of a vector y by a scalar may be accomplished with

```
>> z = 3.0*y
```

The transpose, $B = A^T$, and the conjugate transpose, $C = A^H$, are obtained as follows

```
>> B = A.', C=A'
```

The inverse of a square matrix (if it exists) may be obtained by typing

```
>> D = inv(A)
```

This command is rewritten in §2 of the present text (see Algorithm 2.3); as mentioned in §2 (indeed, as identified as early as §1.2.7), you should not ever compute a matrix inverse (which is expensive to calculate) in a production code, though it is sometimes convenient to compute a matrix inverse in a test code.

A 5×5 identity matrix may be constructed with

```
>> E = eye(5)
```

Tridiagonal matrices may be constructed by the following command and variations thereof:

```
>> 1*diag(ones(m-1,1),-1) - 2*diag(ones(m,1),0) + 1*diag(ones(m-1,1),1)
```

See also Algorithm 1.1. There are two “matrix division” symbols in Matlab/Octave, \ and / — if A is a nonsingular square matrix, then A\B and B/A correspond formally to left and right multiplication of B (which must be of the appropriate size) by the inverse of A, that is $\text{inv}(A) * B$ and $B * \text{inv}(A)$, but the result is obtained directly (via Gaussian elimination with complete pivoting, as discussed, and rewritten from scratch, in §2, but leaving the matrix A in tact) without the computation of the inverse (which is a very expensive computation). Thus, to solve a system $A * x = b$ for the unknown vector x, one may type

```
>> A=[1 2 3; 4 5 6; 7 8 0]; b=[5 8 -7]'; x=A\b
```

which results in

```
x =
    -1
     0
     2
```

To check this result, just type

```
>> A*x
```

which verifies that

```
ans =
     5
     8
    -7
```

Starting with the innermost group(s) of operations nested in parentheses and working outward, the usual precedence rules are observed by Matlab/Octave. First, all the exponentials are calculated. Then, all the multiplications and divisions are calculated. Finally, all the additions and subtractions are calculated. In each of these three categories, the calculation proceeds from left to right through the expression. Thus

```
>> 5/5*3
ans = 3
```

and

```
>> 5/(5*3)
ans = 0.3333
```

If in doubt, use parentheses to ensure the order of operations is as you intend. Note that the matrix sizes must be correct for the requested operations to be defined or an error will result.

Suppose we have two vectors **x** and **y** and wish to perform the componentwise operations:

$$z_{\alpha} = x_{\alpha} * y_{\alpha} \quad \text{for } \alpha = 1, n.$$

The Matlab/Octave command to execute this is

```
>> x=[1:5]'; y=[6:10]'; z=x.*y
```

Note that `z=x*y` gives an error, since this implies matrix multiplication and is undefined in this case. (A row vector times a column vector, however, is a well defined operation, so `z = x'*y` is successful. Try it!) The period distinguishes matrix operations (`*`, `^` and `/`) from component-wise operations (`.*`, `.^` and `./`).

Typing `whos` lists all the variables created up to that point, and typing `clear` removes them. The `format` command is also useful for changing how Matlab/Octave prints things on the screen (type `help format` for more information). In order to save the entire variable set computed in a session, type `save session` prior to quitting. At a later time the session may be resumed by typing `load session`.

Some additional (self-explanatory) functions include: `abs`, `conj`, `real`, `imag`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `exp`, `log`, `log10`. Some useful predefined constants include `pi`, `i`, `eps`; note that your code can change the values of these constants (it is particularly common to use `i` as an indexing variable) — be especially careful if you decide to do this!

Matlab and Octave are also distributed with many special additional functions to aid in linear problem-solving, control design, etc. In Octave (but, unfortunately, not in recent versions of Matlab), many of these advanced built-in functions are themselves saved as prewritten `m`-files (see below), and can easily be opened and accessed by the user for examination. *The present pedagogical text specifically avoids using most of these convenient **black-box** functions*, instead opting to work up many of them from scratch in order to remove the mystery that might otherwise be associated with using them.

Note that, in order to make a Octave **flush** (that is, print messages to the screen) before it either finishes running or encounters a `pause` statement in the code, you must insert the command `fflush(1)`; before each `pause` command; such a flush command is not needed in Matlab. This is one of the few yet sometimes annoying differences between Octave and Matlab.

Matlab programming procedures: stay organized!

As an alternative to interactive mode, you can also save a series of Matlab/Octave commands as `m`-files, which are **ASCII (American Standard Code for Information Interchange, a.k.a. plain text)** files, with a descriptive filename ending in `.m`, containing a sequence of commands listed exactly as you would enter them if running interactively. When working on Matlab/Octave problems that take more than one line to complete (that is, essentially, all the time), *it is imperative to write and run `m`-files rather than working in interactive mode*. By so doing, it is self evident which calculation followed from which. Further, following this approach, the several commands typically required to perform a given calculation don't need to be retyped when the simulation needs to be rerun, which is generally more often than one would care to admit. *Staying organized with different versions of your `m`-files as the project evolves is essential; create new directories and subdirectories as appropriate for each problem you work on to stay organized, and to keep from accidentally overwriting previously-written and debugged codes.*

To execute the commands in an `m`-file named⁶ `foo.m`, simply type `foo` at the Matlab/Octave prompt. Any text editor may be used to edit `m`-files. The `%` symbol is used in these files to indicate that the rest of that line is a comment. Entering `help foo` will print the first (commented) lines of the code to the screen, entering `type foo` will print the entire code to the screen⁷. *Comment all `m`-files clearly, sufficiently, and succinctly*, so that you can come back to the code later and understand how it works. Depending on the purpose of the code, you can also print to the screen (using the `disp` command) to update you on the code's progress as it runs; several codes presented in this text use a `verbose` flag to turn such updates on or off. It also helps to use descriptive variable names to indicate what the code is doing.

⁶Almost all texts describing computer programming make reference to simple example codes named `foo` and `bar`, though the etymology of this tradition is not precisely known.

⁷In the electronic version of this text, click [View](#) to see a plain text version of the production `m`-file in a web browser, and click [Test](#) to see the corresponding test code. To save space, the copyleft comments at the beginning of each code are not printed in the text.

The distinction between functions and scripts

There are two types of `m`-files: **scripts** and **functions**. A **script** is a set of Matlab/Octave commands which run exactly as if you had typed in each command interactively. A script has access to all previously-defined variables (that is, it inherits the **global workspace**).

A **function**, on the other hand, is a set of Matlab/Octave commands that begins with a `function` declaration, for example,

```
function [o1,o2] = bar(i1,i2,i3)
```

A function so defined may then be **called** (as in a compiler-based programming language) with the command

```
[c,d] = bar(a,b,c)
```

Note that some variables (in the above example, `c`) may be used as both inputs and outputs. As a function is running, it can only reference those variables in the input list with which it was called; in the present example, the function is only “aware” of the variables `a`, `b`, and `c`, which the function refers to internally as `i1`, `i2`, and `i3`. Conversely after the function is finished, the only variables that are modified as compared with before the function was called are those variables in the output list with which it was called; in the present example, the function ultimately only modifies the variables `c` and `d`, which the function refers to internally as `o1` and `o2`.

The special variables `nargin` and `nargout` identify the number of input and output arguments, respectively, that are used when any given function is called. Input and output arguments are assigned from the left to the right, so any missing variables are necessarily those at the end of each list. If some of the input arguments are omitted in the function call, these variables may be set to default values by the function. If some of the output arguments are omitted in the function call, the function may sometimes be accelerated by avoiding the explicit computation of these variables.

Functions are more easily extended to other problems than scripts, as functions make clear via their argument list what information from the calling function is used in, and returned by, the called function⁸. In complicated codes, unintentionally assigning a minor variables (like the index `i`) with different meanings in different scripts that call each other can lead to a bug that is very difficult to find. The proper use of functions, and the associated passing of only the relevant data back and forth (known as **handshaking**), goes a long way towards preventing such bugs from appearing in your production codes.

On the other hand, short test codes, such as those provided with each of the functions developed in this text, are usually convenient to write as scripts, so that the variables defined by the test script may be checked (for debugging purposes) after the test script runs. To illustrate, ten simple functions and one associated test script are listed in Algorithm A.1; in the comments at the beginning of each function is a brief description of what that function does. The coefficients of polynomials are represented frequently in this text as row vectors; the functions listed in Algorithm A.1 are useful to perform polynomial addition, convolution, division, etc. Algorithm A.2 is an example script illustrating the use of Matlab’s built-in functions `eig` and `inv` to build matrices of various structure, as discussed further in §1 and 4.

The primary directory Matlab uses (for both saving new files and reading old ones) may be changed with the command `cd <directory name>`; in Matlab, there are also menu-driven ways of accomplishing this. If the necessary `m`-files to run your code are stored in more than one directory, which is often the case, the `path` command may be used to indicate to Matlab these additional directories in which to look for the appropriate `m`-files. A convenient startup script is given in Algorithm A.3 to extend the Matlab/Octave path to access all of the codes of the *Numerical Renaissance Codebase (NRC)*; it is convenient (and, thus, recommended) to place a copy of this file, named `startup.m`, in the directory where Matlab/Octave normally starts up on your computer in order to initialize this path on startup (note that the variable `base` in this `startup.m` file must be updated appropriately in order to point at the directory where the *NRC* is stored on your computer).

⁸That is, in addition to those variables defined as `global`, as seen in Algorithm 11.8 and discussed in footnote 22 on page 359.

Algorithm A.1: Some simple example functions, and an example test script.

<pre>function p=Prod(a) % Compute the product p of the elements in the vector a. p=a(1); for i=2:length(a); p=p*a(i); end end % function Prod</pre>	View
<pre>% script ProdTest disp('Compute the product of the elements in a'), a=[2 4 5], p=Prod(a), disp(' ') % end script ProdTest</pre>	View
<pre>function a=Fac(b) % Compute the factorial of each element of the matrix b. [n,m]=size(b); for i=1:n,for j=1:m, a(i,j)=1; for k=2:b(i,j); a(i,j)=a(i,j)*k; end,end,end end % function Fac</pre>	View Test
<pre>function p=Poly(r) % Compute the coefficients of the polynomial with roots r. n=length(r); p=1; for i=1:n; p=PolyConv(p,[1 -r(i)]); end end % function Poly</pre>	View Test
<pre>function v=PolyVal(p,s) % For n=length(p), compute p(1)*s(i)^(n-1) + p(2)*s(i)^(n-1) + ... + p(n-1)*s(i) + p(n) % for each s(i) in the vector s. n=length(p); for j=1:length(s); v(j)=0; for i=0:n-1, v(j)=v(j)+p(n-i)*s(j)^i; end, end end % function PolyVal</pre>	View Test
<pre>function [sum]=PolyAdd(a,b,justification) % Add row vectors a and b with either 'r' (default) or 'l' justification. m=length(a); n=length(b); if nargin<3, justification='r'; end if justification=='r', a=[zeros(1,n-m) a]; b=[zeros(1,m-n) b]; else a=[a zeros(1,n-m)]; b=[b zeros(1,m-n)]; end, sum=a+b; end % function PolyAdd</pre>	View Test
<pre>function p=PolyConv(a,b,c,d,e,f,g,h,i,j) % Recursively compute the convolution of the two to ten polynomials, given as arguments. if nargin>9, e=PolyConv(e,j); end, if nargin>8, e=PolyConv(e,i); end if nargin>7, e=PolyConv(e,h); end, if nargin>6, e=PolyConv(e,g); end if nargin>5, e=PolyConv(e,f); end, if nargin>4, d=PolyConv(d,e); end if nargin>3, c=PolyConv(c,d); end, if nargin>2, b=PolyConv(b,c); end m=length(a); n=length(b); p=zeros(1,n+m-1); for k=0:n-1; p=p+[zeros(1,n-1-k) b(n-k)*a zeros(1,k)]; end end % function PolyConv</pre>	View Test
<pre>function [d,r]=PolyDivide(num,den) % Perform polynomial division of den into num, resulting in div with remainder rem. m=length(num); n=length(den); r=num; d=0; for j=1:m-n+1, d(j)=r(1)/den(1); r=PolyAdd(r,-d(j)*den,'l'); r=r(2:end); end end % function PolyDivide</pre>	View Test
<pre>function b=PolyPower(p,n) % Compute the convolution of the polynomial p with itself n times. if n==0, b=1; else, b=p; for i=2:n, b=PolyConv(b,p); end end % function PolyPower</pre>	View Test
<pre>function p=PolyDiff(p,d,n) % Recursively compute the d'th derivative of a polynomial p of order n. if nargin<2, d=1; end, if nargin<3, n=length(p)-1; end if d>0, p=[n:-1:1].*p(1:n); if d>1, p=PolyDiff(p,d-1,n-1); end, end end % function PolyDiff</pre>	View Test
<pre>function [b,a]=Swap(a,b) % A curiously simple (empty!) function that swaps the contents of a and b. end % function Swap</pre>	View Test

Algorithm A.2: An m-file, written as a script, illustrating the use of Matlab's built-in functions `eig` and `inv`.

View

```
% script <a href="matlab:SampleMatlabBuiltInFns">SampleMatlabBuiltInFns </a>
% This sample script finds the eigenvalues and eigenvectors of a few random matrices
% constructed with special structure. This sample script uses Matlab's built-in commands
% <a href="matlab:doc inv">inv</a> and <a href="matlab:doc eig">eig</a> (lower case).
% NR develops, from first principles, alternatives to matlab's built-in commands, such as
% <a href="matlab:help Inv">Inv</a> and <a href="matlab:help Eig">Eig</a> (mixed case).
clear, R = randn(4), eigenvalues = eig(R)
disp('As R has random real entries, it may have real or complex eigenvalues.')
```

```
disp('Press return to continue'), pause, R_plus_Rprime = R+R', eigenvalues = eig(R+R')
disp('Notice that R+R' is symmetric, with real eigenvalues.')
```

```
disp('Press return to continue'), pause, R_minus_Rprime = R-R', eigenvalues = eig(R-R')
disp('The matrix R-R' is skew-symmetric, with pure imaginary eigenvalues.')
```

```
disp('Press return to continue'), pause, Rprime_times_R = R'*R, [S,Lambda] = eig(R'*R)
disp('This matrix R'*R is symmetric, with real POSITIVE eigenvalues.')
```

```
disp('Press return to continue'), pause, R_times_Rprime = R*R', [S,Lambda] = eig(R*R')
disp('R*R' has the same eigenvalues as R'*R, but different eigenvectors.')
```

```
disp('Press return to continue'), pause, S = rand(4); Lambda = diag([1 2 3 4]);
A = S * Lambda * inv(S), eigenvalues_of_A = eig(A)
disp('You can also create a matrix with desired eigenvalues (say, 1, 2, 3, 4)')
```

```
disp('from A=S Lambda inv(S) for any invertible S.'), disp(' ')
% end script SampleMatlabBuiltInFns
```

Algorithm A.3: A script to extend the path of Matlab/Octave to include the Numerical Renaissance Codebase.

View

```
% script <a href="matlab:NRCpathsetup">NRCpathsetup </a>
% Initialize the path environment for using the Numerical Renaissance Codebase.
% Tip: set up a symbolic link in a convenient place to make it easy to call this script
% when firing up matlab or octave. This can be done, e.g., in Mac OS X as follows:
% ln -s /usr/local/lib/NRC/NRchapAB/NRCpathsetup.m ~/Documents/MATLAB/startup.m
% Be sure to modify "base" appropriately below if the NRC library is not in /usr/local/lib
base='/usr/local/lib/NRC/'; format compact, clc, close all, cd ~
addpath(strcat(base,'NRchap01'),strcat(base,'NRchap02'),strcat(base,'NRchap03'), ...
        strcat(base,'NRchap04'),strcat(base,'NRchap05'),strcat(base,'NRchap06'), ...
        strcat(base,'NRchap07'),strcat(base,'NRchap08'),strcat(base,'NRchap09'), ...
        strcat(base,'NRchap10'),strcat(base,'NRchap11'),strcat(base,'NRchap12'), ...
        strcat(base,'NRchap13'),strcat(base,'NRchap14'),strcat(base,'NRchap15'), ...
        strcat(base,'NRchap16'),strcat(base,'NRchap17'),strcat(base,'NRchap18'), ...
        strcat(base,'NRchap19'),strcat(base,'NRchap20'),strcat(base,'NRchap21'), ...
        strcat(base,'NRchap22'),strcat(base,'NRchapAA'),strcat(base,'NRchapAB'), ...
        strcat(base,'NRextra'),strcat(base,'NRextra/exportfig'),base)
disp([' Path set for using Numerical Renaissance Codebase; ' ...
      'type <a href="matlab:help NRC">help NRC</a> to get started.' char(10)])
% end script NRCpathsetup
```

The function `SuDokuSolve` solves the well-known SuDoku problem. Note that Algorithm A.4 defines the auxiliary functions `PlaySuDoku`, `PrintSuDoku`, and `RecursiveSuDoku` that may only be executed when the main function `SuDokuSolve` is running; if it is desired to make these codes accessible outside of the function `SuDokuSolve`, they should each be saved in their own m-file.

The overhead associated with function calls

There is extra overhead associated with function calls, because they often **allocate** (that is, assign) new memory locations for variables passed in when called, then **deallocate** (that is, release) this memory upon exit. This behavior is referred to as **passing by value**. For small variables, the overhead associated with passing by value is minimal. For large arrays, however such overhead can be substantial. To avoid this overhead, one

may either use globally-defined arrays, as done in Algorithms 11.8-11.10, or **pass by reference**, which means to pass a **pointer** to the original memory location of the array rather than copy the values in the array into a new memory location. This is the default for array passing in Fortran and is common in C. Matlab's default is essentially a pass-by-value approach, with memory allocation deferred to the first time the array is modified within the subprogram. However, a pass-by-reference mode is initiated in Matlab whenever a function calls another function with an identical argument in both the input and output lists (in both the calling function and the called function); this can lead to significant improvements in execution speed for large problems.

When passing several parameters back and forth between functions, it is often convenient to group these parameters as derived data types, as illustrated, for example, in Algorithm 10.3.

Algorithm A.4: A more complicated function that solves the SuDoku problem.

```
function SuDokuSolve(F)
% An involved recursive code for solving SuDoku problems, emulating how a human plays.
% First, split SuDoku 9x9 array into a 3x3x3x3 array, which is easier for analysis
for i=1:3; for j=1:3; for k=1:3; for l=1:3; A(i,j,k,l)=F(i+(k-1)*3,j+(l-1)*3); end; end; end; end
PrintSuDoku(A), [A,B,flag]=PlaySuDoku(A);
PrintSuDoku(A), if flag==0; [A]=RecursiveSuDoku(A,B,flag); PrintSuDoku(A); end
end % function SuDokuSolve
function [A,B,flag]=PlaySuDoku(A)
% This routine attempts to solve the SuDoku puzzle directly (without guessing).
% On exit, flag=1 indicates that a unique solution has been found,
% flag=0 indicates uncertainty (i.e., not enough information to solve without guessing),
% flag=-1 indicates failure (i.e., the data provided is inconsistent).
B=ones(3,3,3,3,9); flag=0; % B keeps track of all possible values of the unknown entries.
M=1; while M==1; M=0;
    % M=1 means something has been determined this iteration, and we need to iterate again.
    for i=1:3; for j=1:3; for k=1:3; for l=1:3; % Fill B with A
        if A(i,j,k,l)>0; B(i,j,k,l,:)=0; B(i,j,k,l,A(i,j,k,l))=1; end
    end; end; end; end
    for i=1:3; for j=1:3; for k=1:3; for l=1:3; for m=1:3; for n=1:3; % Reduce B checking
        if ((m~=j) | (n~=l)) & A(i,m,k,n)>0; B(i,j,k,l,A(i,m,k,n))=0; end % ... row (i,k)
        if ((m~=i) | (n~=k)) & A(m,j,n,l)>0; B(i,j,k,l,A(m,j,n,l))=0; end % ... column (j,l)
        if ((m~=i) | (n~=j)) & A(m,n,k,l)>0; B(i,j,k,l,A(m,n,k,l))=0; end % ... square (k,l)
    end; end; end; end; end
    ME=1; while ME==1; ME=0; for i=1:3; for j=1:3; for k=1:3; for l=1:3; % Check each element
        E=sum(B(i,j,k,l,:));
        if E==0; flag=-1; return; elseif (E==1) & (A(i,j,k,l)==0);
            ME=1; M=1; [y,A(i,j,k,l)]=max(B(i,j,k,l,:));
        end
    end; end; end; end; end
    MR=1; while MR==1; MR=0; for i=1:3; for k=1:3; for m=1:9; % Check each row
        R=sum(sum(B(i,:,k,:),m));
        if R==0; flag=-1; return; elseif R==1;
            [y1,jv]=max(B(i,:,k,:),m,[],2); [y1,l]=max(y1,[],4);
            if A(i,jv(1),k,l)==0; A(i,jv(1),k,l)=m; MR=1; M=1; end;
        end
    end; end; end; end
    MC=1; while MC==1; MC=0; for j=1:3; for l=1:3; for m=1:9; % Check each column
        C=sum(sum(B(:,j,:,l,m)));
        if C==0; flag=-1; return; elseif C==1;
            [y2,iv]=max(B(:,j,:,l,m,[],1); [y2,k]=max(y2,[],3);
            if A(iv(k),j,k,l)==0; A(iv(k),j,k,l)=m; MC=1; M=1; end;
        end
    end; end; end; end
    MS=1; while MS==1; MS=0; for k=1:3; for l=1:3; for m=1:9; % Check each square
        S=sum(sum(sum(B(:, :, k, l, m))));
        if S==0; flag=-1; return; elseif S==1;
            [y3,iv]=max(B(:, :, k, l, m), [], 1); [y3,j]=max(y3, [], 2);
```

View
Test

```

        if A(iv(j),j,k,l)==0; A(iv(j),j,k,l)=m; MS=1; M=1; end;
    end
end; end; end; end
end
if (min(min(min(min(A))))==1) flag=1; end;
end % function PlaySuDoku
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function PrintSuDoku(A)
A=[A(:, :, 1, 1) A(:, :, 1, 2) A(:, :, 1, 3); A(:, :, 2, 1) A(:, :, 2, 2) A(:, :, 2, 3); ...
    A(:, :, 3, 1) A(:, :, 3, 2) A(:, :, 3, 3)]
end % function PrintSuDoku
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [A]=RecursiveSuDoku(A,B,flag)
% If the previous call to PlaySuDoku is inconclusive, then this routine coordinates
% recursively one or more guesses until a solution (which might not be unique) is found.
for i=1:3; for j=1:3; for k=1:3; for l=1:3;
    Asave=A; Bsave=B;
    if A(i,j,k,l)==0; for m=1:9; if B(i,j,k,l,m)==1;
        disp(sprintf(' Guessing  A(%d,%d)=%d', i+(k-1)*3, j+(l-1)*3, m))
        A(i,j,k,l)=m; [A,B,flag]=PlaySuDoku(A);
        if flag==-1; disp('failure'); A=Asave; B=BSave;
        elseif flag==0; disp('uncertain'); [A]=RecursiveSuDoku(A,B,flag); return;
        else; disp('DONE!'); return
    end
end; end; end; end
end; end; end; end
end % function RecursiveSuDoku

```

Plotting

Both 2D and 3D plots are easy to generate in Matlab and Octave, as shown below:

A sample 2D plot

```

x=[0:.1:10];
y1=sin(x);
y2=cos(x);
plot(x,y1,'-',x,y2,'x')

```

A sample 3D plot

```

[x,y] = meshgrid(-8:.5:8,-8:.5:8);
R = sqrt(x.^2 + y.^2) + eps;
Z = sin(R)./R;
mesh(x,y,Z)

```

The code segments listed above produce the figures shown in Figure A.1. Note that axis rescaling and labeling can be controlled with `loglog`, `semilogx`, `semilogy`, `title`, `xlabel`, `ylabel`, etc. (see the help pages on these and related commands for more information).

The plotting commands illustrated above produce plots in figure windows. Once your code is working correctly and the plot is as you like it, you will usually want to save it, so you can email it, make printouts of it, and/or include it in a report or paper you are writing. For such reports and papers, \LaTeX invariably produces the best results, though you might find that **what-you-see-is-what-you-get (wysiwyg)** word processors such as Microsoft Word are, initially, somewhat easier to use⁹. The recommended format to save your figures for such purposes is **encapsulated postscript** (typically denoted with a **.eps** suffix). Encapsulated postscript is a robust, platform-independent standard for graphics files that saves lines as vectors (lines) rather than as bitmaps (pixels), and therefore looks sharp when printed. All major typesetting programs, including \LaTeX and Microsoft Word, can import .eps files.

To produce a color .eps file, type the following command after executing the plotting commands:

⁹That is, until you begin to care about how well the equations are typeset, at which point your best option (by far, in the opinion of the author) is to abandon Microsoft Word and switch to \LaTeX ...

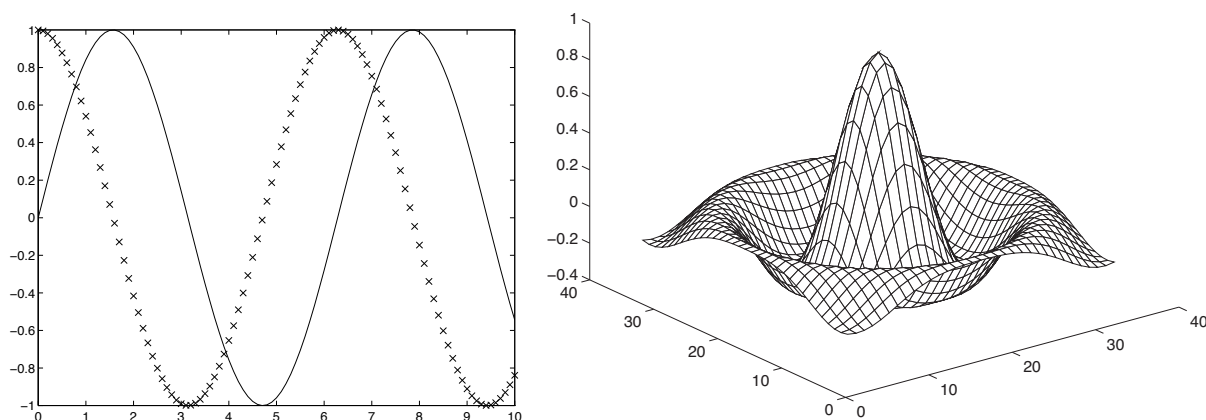


Figure A.1: Sample 2D and 3D plots.

```
print -depsc foo.eps
```

Once you have created an .eps file, you may view it with a free piece of software called **Ghostview (gv)**. **Adobe Illustrator** is a very useful commercial software package that may be used to further edit .eps files, which is often necessary when preparing scientific papers.

In Matlab, the contents of a figure window may also be saved with the command `saveas(1, 'foo.fig')`, and later reopened and edited further with the command `open foo.fig`; this command does not exist in Octave. This command is not entirely recommended, however, because .fig files are often not portable from one platform to another. Instead, to send a figure via email, it is recommended that you create a .eps file (as described above) and send that. If you want the option to edit the figure later in Matlab or Octave, your flexibility is maximized if you save in an m-file the list of commands that created the figure (or, alternatively, save the session, as discussed previously) and recreate the figure from scratch later.

Note also that printouts of the text appearing in the Matlab or Octave command window after a code is run is best achieved simply by copying this text and pasting it in to the editor of your choosing, then printing from there.

Advanced prepackaged numerical routines

Matlab and Octave have many many advanced prepackaged numerical routines built in. For example, a few prepackaged linear algebra commands which you might find useful (see the respective help pages for details) include: `lu`, `inv`, `hess`, `qr`, `orth`, `schur`, `eig`, `jordan`, `svd`, `chol`, `trace`, `norm`, `cond`, `pinv`, etc. These routines will certainly be useful for you as you learn how they work and why they are useful. However, the purpose of this text is *not* simply to catalog how to use these prepackaged routines (there are plenty of other texts that accomplish that), but rather to flush out, for some of the most important of these routines, *how* they actually work, and *why* they are useful. With this knowledge, the reader will be able to select and use these routines with much greater understanding and forethought. We thus, as mentioned previously, explicitly and intentionally avoid using *all* such advanced prepackaged routines in this text, instead learning to write these routines from scratch ourselves.

Exercises & References

The only reference required to get up to speed in both Matlab and Octave is the online help, which is quite extensive. Programming is mostly an exercise in logical organization and is best learnt by example; by following through the many examples and exercises laid out in this text, you will quickly become proficient at writing clear, effective, efficient, and reusable numerical codes.