

A Beginner's Guide to MATLAB

CONTENTS

1. Introduction	2
2. Basic Matlab Commands	2
2.1 How to run Matlab	2
2.2 Matrices	3
2.3 Matrix Operatons	4
2.4 Matrix Buliding Functions	5
2.5 Scalar, Vector and Matrix Functions	6
2.6 Colon Operator and Submatrices	7
2.7 Plotting and Graphics	8
3. Programming in Matlab	10
3.1 Relational and Logical Operators	10
3.2 Loops and Conditional Structures	10
3.3 Matlab Scripts (M-files)	12
3.4 Matlab Function	12
3.5 Debugging a Matlab M-file	14
3.6 Programming Tips	15
4. References	18

A Beginner's Guide to MATLAB

1. Introduction

MATLAB stands for "MATrix LABoratory" which is an interactive, matrix-based computer program for scientific and engineering numeric computation and visualization. It intended to provide easy access to the matrix libraries developed by *Linpac* and *Eispac* projects. These are carefully tested high-quality programming packages for solving linear equations and eigenvalue problems.

MATLAB will be used extensively in the labs of "Signals and Systems" and "Digital Signal Processing" courses. This beginner's guide provides an overview of Matlab, some of its capabilities and the resource of Matlab in the EE department. Matlab is available on a number of computing platforms such as Sun/HP/VAX workstations, 80x86 PCs, Apple Macintosh, and several parallel machines. In the CityU Electronic Engineering Department, Unix version of Matlab 5.0 is available in the Image Processing Lab (P1615) and EDA center. The PC version of Matlab 5.1 is available in Microprocessor Lab (Room 2380). The information of this guide generally applies to all these environments.

The aim of Matlab is to enable us to solve complex numerical problems, without having to write programs in traditional languages like *C* and *Fortran*. Thus, Matlab interprets commands like *Basic* does, instead of compiling source code like *C* and *Fortran* require. By using the relatively simple programming capability of Matlab, it is very easy to create new commands and functions in Matlab. In addition, these developed Matlab programs (or scripts) can run without modification on different computers with Matlab. Today, Matlab has evolved into a very powerful programming environment by providing numerous toolboxes such as signal processing, image processing, controls, optimization, and statistics computations. The emphasis of this beginner's guide is on the basic Matlab commands with investigation on some aspects of signal processing. It is intend to be used, while sitting at a computer terminal running Matlab.

2. Basic Matlab Commands

The Matlab functions are command driven; hence, you have to know the various commands that are available. Because on-line help is available and the syntax structures of most commands are very similar, there is no need to memorize the commands. By building your own expertise on Matlab, you will able to attack numerical problems in engineering quickly.

2.1 How to run Matlab

Unix version of Matlab 5.0 is available at the Image Processing Lab and EDA Center of the Department of Electronic Engineering at City University of Hong Kong. To start the program type matlab at the Unix prompt, as follows:

```
% matlab
```

The PC version of Matlab 5.1 is available at the Microprocessor Lab. To start the PC Matlab, you can select Matlab from the "Program Menu" under Window 95 or Window NT environment.

To exit the Matlab, try the following commands under Matlab prompt:

```
>> exit
or >> quit
```

After enter the Matlab, you can get on-screen information about the Matlab commands with help. Typing

```
>> help
```

produces a listing of all HELP topics and Matlab commands (includes commands related to all the toolboxes available on the path). If you know the name of the function you want help on, you can type:

```
>> help topic
or >> help function_name
```

In addition, you can learn more features of Matlab by running the demo:

```
>> demo
```

A list of demos will appear that will demonstrate how to use Matlab and will show off some of its features and capabilities.

2.2 Matrices

Matlab has only one data type: a complex-valued floating-point matrix. A vector is simply a matrix with either one row [a row vector] or one column [a column vector]. A number, or scalar, is simply a 1-by-1 matrix.

Variables in Matlab must start with an alphabetic character, and must contain only alphabetic characters, numeric characters, and the underscore character, e.g. **data** is okay, but **data!** is not. Matlab is also case sensitive, e.g. **data** and **Data** are not the same identifier.

For example, enter the following vectors and matrix in Matlab as

```
>> rowvector = [1 2 3]
rowvector =
     1     2     3

>> columnvector = [1; 2; 3]
columnvector =
     1
     2
     3

>> matrix2x2 = [1 2; 3 4]
matrix2x2 =
     1     2
     3     4
```

In the matrix addressing and subscripting, Matlab is similar enough to *C* to cause some extremely annoying errors. One such error is that array and vector indexing begins with 1 and not 0.

```
>> x = [1 3 5 7 9]
x =
     1     3     5     7     9
```

```
>> x(0)
??? Index exceeds matrix dimensions.
>> x(1)
ans =
    1
```

For example, a 3-by-3 matrix **A** is defined as

```
>> A = [1,2,3; 4,5,6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
```

The element in the *i*'th row and *j*'th column of **A** is referred to in the usual way:

```
>> A(2,3)
ans =
     6
```

In this example the expression was not explicitly assigned to a variable so Matlab automatically assigned the result to the variable **ans**. It's very easy to modify matrices:

```
>> A(3,2) = 10
A =
     1     2     3
     3     5     6
     7    10     9
```

2.3 Matrix and Array Operations

Matlab supports the following matrix arithmetic operations:

+	addition	-	subtraction
*	multiplication	^	power
\	left division	/	right division
'	transpose		

To experience the matrix arithmetic in the Matlab, enter the following **A** and **b** matrices

```
>> A = [1 2 3 ; 4 5 6 ; 7 8 10], b = [2; 4; 6]
```

The transpose of a matrix is the result of interchanging rows and columns. Matlab denotes the [conjugate] transpose by following the matrix with the single-quote [apostrophe].

```
>> B = A'
B =
     1     4     7
     2     5     8
     3     6    10
```

Scalars multiply matrices as expected, and matrices may be added in the usual way; both are done "element by element."

```
>> 2*A
ans =
     2     4     6
     8    10    12
    14    16    18
```

Also try the following two commands:

```
>> A/3
>> A + [b,b,b]
```

Scalars added to matrices produce a "strange" result, but one that is sometimes useful; the scalar is added to every element.

```
>> A+1
ans =
     2     3     4
     5     6     7
     8     9    10
```

Matrix multiplication requires that the sizes match. If they don't, an error message is generated.

```
>> A*b
>> b'*A
>> A*A', A'*A
>> b'*b, b*b'
```

To perform point-wise operation (array operation) on matrices, we can use the "point-star" operator, e.g. $\mathbf{A} \cdot * \mathbf{B}$. In general, where "point" is used with another arithmetic operator it modifies that operator's usual matrix definition to a point-wise one. Thus we have $\cdot /$ and $\cdot ^$ for point-wise division and exponentiation.

Try the following commands to gain more experience on point-wise operations.

```
>> A^2, A.^2
>> A.*A, b.*b
>> 1./A
>> 1./A.^2
```

2.4 Matrix Building Functions

Matlab has a number of matrix building functions such as:

<code>eye</code>	Identity matrix
<code>zeros</code>	Matrix of zeros
<code>ones</code>	Matrix of ones
<code>diag</code>	Diagonal matrix
<code>hilb</code>	Hilbert matrix
<code>magic</code>	Magic square
<code>triu</code>	Upper triangular part of a matrix
<code>tril</code>	Lower triangular part of a matrix
<code>rand</code>	Randomly generated matrix

For example, `zeros(m,n)` produces an m -by- n matrix of `zeros` and `zeros(n)` produces an $n \times n$ zero matrix. In addition, if \mathbf{A} is a m -by- n matrix, then `zeros(A)` produces a matrix of zeros of the same size as \mathbf{A} . In our example \mathbf{A} is 3-by-3 matrix, so

```
>> zeros(A)
ans =
     0     0     0
     0     0     0
     0     0     0
```

If **b** is a vector, **diag(b)** is the diagonal matrix with **b** down the diagonal; if **A** is a square matrix, then **diag(A)** is a vector consisting of the diagonal of **A**. What is **diag(diag(A))**? Try it.

Matrices can be built from blocks. For example, if **A** is a 3-by-3 matrix, then

```
B = [A, zeros(3,2); zeros(2,3), eye(2)]
```

will build a certain 5-by-5 matrix. Try it.

2.5 Scalar, Vector and Matrix Functions

Matlab has some functions operate essentially on scalars while operate element-by-element when applied to a matrix. The most common scalar functions are

sin	cos	tan
asin	acos	atan
exp	sign	rem (remainder)
abs	sqrt	log (natural log)
round	floor	ceil

Another useful Matlab functions are vector-oriented which operate on vectors (row or column matrices). If they are applied to a matrix, they are computed on a column-by-column basis. Some of these functions are:

max	min	prod
median	mean	std
sum	cumsum	sumprod

Matlab supports the paradigm of "functional programming" which in the language of system theory is equivalent to cascading systems. For example, to find the largest element in the Matrix **A**, we can cascade the command as **max(max(A))** such as,

```
>> max(max(A))
ans =
    9
```

For the commands **max** and **min** also return the corresponding index (or location) of the minimum or maximum value, respectively, when with two output arguments. For example,

```
>> [Max_b index_b] = max(b)
Max_b =
    6
index_b =
    3
```

The most powerful feature of the Matlab is its matrix functions. The most useful ones are

inv	- Matrix inverse.
det	- Determinant.
norm	- Matrix or vector norm.
size	- Size of the matrix.
rank	- Number of linearly independent rows or columns.
eig	- Eigenvalues and eigenvectors.
svd	- Singular value decomposition.
poly	- Characteristic polynomial.
cond	- Matrix condition number.
expm	- Matrix exponential.

```

logm      - Matrix logarithm.
sqrtm     - Matrix square root.
funm      - Evaluate general matrix function.
lu        - Factors from Gaussian elimination.
qr        - Orthogonal-triangular decomposition.

```

For example, **A** is a 3-by-3 matrix, the determinant, rank, nullspace, and condition number.

```

>> det(A)
>> rank(A)
>> norm(A)
>> null(A)

```

2.6 Colon Operator and Submatrices

The colon operator is very useful for creating index arrays, creating vectors of evenly spaced values, and accessing submatrices. The help for `:` is given below:

```

>>help colon
:      Colon. Used in subscripts, FOR iterations and elsewhere.
      J:K is the same as [J, J+1, ..., K]
      J:K is empty if J > K.
      J:I:K is the same as [J, J+I, J+2I, ..., K]
      J:I:K is empty if I > 0 and J > K or if I < 0 and J < K.
      The colon notation can be used to pick out selected rows,
      columns and elements of vectors and matrices.
      A(:) is all the elements of A, regarded as a single
      column. On the left side of an assignment statement, A(:)
      fills A, preserving its shape from before.
      A(:,J) is the J-th column of A
      A(J:K) is A(J),A(J+1),...,A(K)
      A(:,J:K) is A(:,J),A(:,J+1),...,A(:,K) and so on.
      For the use of the colon in the FOR statement, See FOR.

```

The colon notation works from the idea that an index range can be generated by giving a **start**, a **step**, and then the **end**. Therefore, a regularly spaced vector of integers is obtained via

```
iii = start:step:end
```

Without the **step** parameter, the increment is 1. This sort of counting is similar to the notation used in FORTRAN DO loops.

For example, the expression `1:6` is actually a row vector `[1 2 3 4 5 6]`. Try

```

>> 1:6
ans =
     1     2     3     4     5     6

```

The numbers need not be integers. For example,

```

>> 0.1:0.2:0.9
ans =
    0.1000    0.3000    0.5000    0.7000    0.9000

```

Also try

```

>> 6:-1:1
ans =
     6     5     4     3     2     1

```

The following statements will, for example, generate a sinusoidal wave vector \mathbf{y} . Try it.

```
x = [0.0:0.1:2.0]';  
y = sin(x);
```

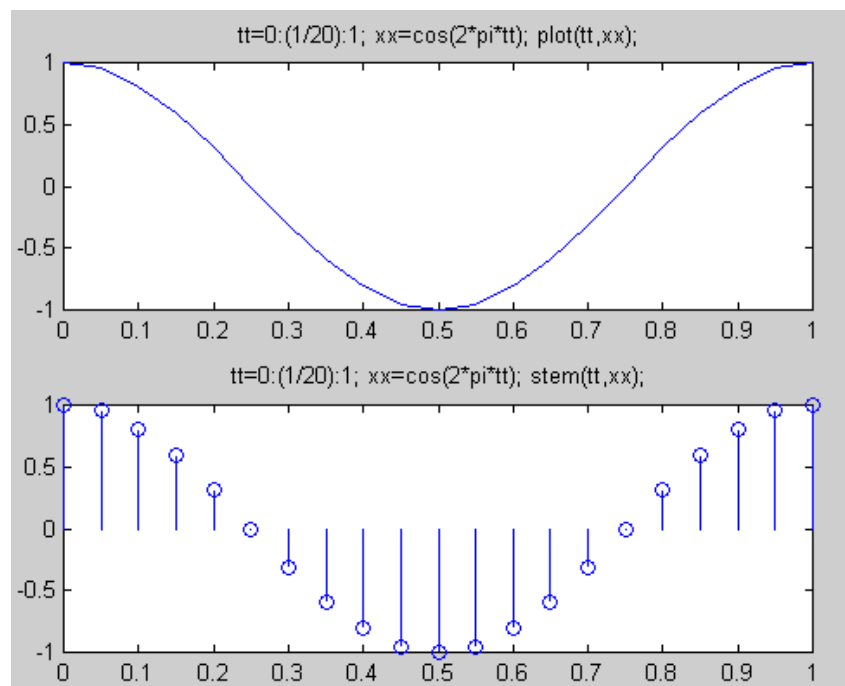
Note that since `sin` operates entry-wise, it produces a vector \mathbf{y} from the vector \mathbf{x} .

The colon notation can be used to access submatrices of a matrix. If you start with the matrix \mathbf{A} , then $\mathbf{A}(2,3)$ is the scalar element located at the 2nd row, and 3rd column of \mathbf{A} . But you can also pull out a 4×3 sub-matrix via $\mathbf{A}(2:5,1:3)$. If you want an entire row, the colon serves as a wild card: i.e., $\mathbf{A}(2,:)$ is the 2nd row. You can even flip a vector by just indexing backwards: $\mathbf{x}(L:-1:1)$. Finally, it is sometimes necessary to just work with all the values in a matrix, so $\mathbf{A}(:)$ gives a column vector that is just the columns of \mathbf{A} concatenated together. More general "reshaping" of the matrix \mathbf{A} can be accomplished with the `reshape(A,M,N)` function. For example, a 5×4 matrix \mathbf{B} can be reshaped into a 2×10 matrix via:

```
Bnew = reshape(B,2,10)
```

2.7 Plotting and Graphics

Matlab is capable of producing 2-D and 3-D plots, displaying images, and even creating and playing movies. The two most common plotting functions that will be used in the Digital Signal Processing are `plot` and `stem`. The basic forms of `plot` and `stem` are the same with `plot(x,y)` producing a connected plot with data points $\{\{x(1),y(1)\}, \{x(2),y(2)\}, \dots, \{x(N),y(N)\}\}$ and `stem` producing a "lollipop" presentation of the same data as show below:



Multiple plots per page can be done with the subplot function. To set up a 3×2 tiling of the figure window, use `subplot(3,2, title_number)`. For example, `subplot(3,2,3)` will direct the next plot to the third tile which is in the second row, left side. For example, the above two figures can be put together in one page using the following commands:

```
>>subplot(2,1,1); plot(tt,xx);
>>subplot(2,1,2); stem(tt,xx);
```

In addition, the command `grid` will place grid lines on the current graph. The graphs can be given title, axes labeled, and text placed within the graph with the following commands which take a string as an argument.

title	graph title
xlabel	x-axis label
y-label	y-axis label
gtext	interactively-positioned text
text	position text at specified coordinates

For example, the command

```
>>title('A sinusoidal wave of 1 Hz');
```

gives a graph a title. The command `gtext('The Spot');` allows a mouse or the arrow keys to position a crosshair on the graph, at which the text will be placed when any key is pressed.

Plots and graphics may be printed to the printer or a file using the `print` command. To send the current plot to default printer simply type `print` without arguments. To save the plot to a file for printing later or including in a document and filename must be specified. For example, a useful format for including the file in a document is encapsulated PostScript, this could be produced as follows:

```
print -deps myplot.eps
```

For a complete list of available file formats, supported printers, and other options see `help print`.

3. Programming in Matlab

Matlab supports some basic programming structures that allow looping and conditioning commands along with relational and logical operators. The syntax and use of some of these structures are very similar to those found in *C*, *Basic*, and *Fortran*. These new commands combined with ones we have discussed earlier can create powerful programs or new functions that can be added to Matlab.

3.1 Relational and Logical Operators

Relational operators allow the comparison of scalars (or matrices, element by element). The result of relational operators is scalars (or matrices of the same size as the arguments) of either 0' or 1's. If the result of comparison is true, the answer is 1; otherwise, it is 0. The following operators are available.

<	- less than	<=	- less than or equal
>	- greater than	>=	- greater than or equal
==	- equal	~=	- not equal

Relations may be connected or quantified by the logical operators

&	- and
	- or
~	- not

When applied to scalars, a relation is actually the scalar 1 or 0 depending on whether the relation is true or false. For example

```
>> 3 > 5
ans =
    0
```

Also try `3 < 5`, `3 == 5`, and `3 == 3`. When the relational and logical operators applied to matrices of the same size, a relation is a matrix of 0's and 1's giving the value of the relation between corresponding entries. For example:

```
>>A = rand(2), B = triu(A), A == B
A =
    0.9103    0.2625
    0.7622    0.0475
B =
    0.9103    0.2625
         0    0.0475
ans =
     1     1
     0     1
```

3.2 Loops and Conditional Structures

Three Matlab commands are available for writing loops, conditional loops, and conditional statements. They are **for**, **while** and **if-else** commands. Basically, these Matlab flow control statements operate like those in most computer languages.

For: For example, for a given **n**, the statement

```
x = []; for i = 1:n, x=[x,i^2], end
```

or

```
x = [];  
for i = 1:n  
    x = [x,i^2]  
end
```

will produce a certain **n**-vector and the statement

```
x = []; for i = n:-1:1, x=[x,i^2], end
```

will produce the same vector in reverse order. Try them. Note that a matrix may be empty (such as **x = []**).

While: The general form of a while loop is

```
while relation  
    statements  
end
```

The statements will be repeatedly executed as long as the relation remains true. For example, for a given number **a**, the following will compute and display the smallest nonnegative integer **n** such that $2^n \geq a$:

```
n = 0;  
while 2^n < a  
    n = n + 1;  
end  
n
```

If: The general form of a simple if statement is

```
if relation  
    statements  
end
```

The statements will be executed only if the relation is true. Multiple branching is also possible, as is illustrated by

```
if n < 0  
    parity = 0;  
elseif rem(n,2) == 0  
    parity = 2;  
else  
    parity = 1;  
end
```

In two-way branching the **elseif** portion would, of course, be omitted.

A relation between matrices is interpreted by while and if to be true if each entry of the relation matrix is nonzero. Hence, if you wish to execute statement when matrices **A** and **B** are equal you could type

```
if A == B  
    statement  
end
```

but if you wish to execute statement when **A** and **B** are not equal, you would type

```
if any(any(A ~= B))
    statement
end
```

or, more simply,

```
if A == B else
    statement
end
```

Note that the seemingly obvious

```
if A ~= B, statement, end
```

will not give what is intended since statement would execute only if each of the corresponding entries of **A** and **B** differ. The functions `any` and `all` can be creatively used to reduce matrix relations to vectors or scalars. Two `any`'s are required above since `any` is a vector operator.

3.3 Matlab Scripts (M-files)

Any expressions which can be entered at the Matlab prompt can also be stored in a text file and executed as a script. The text file can be created with any plain ASCII editor such as *notepad* on a PC and *vi* on UNIX. The file extension must be `.m` and the script is executed in Matlab simply by typing the filename (with or without the extension).

Using your favorite editor, create the following file, named `mymatrix.m`:

```
A = [1 2 3; 4 5 6; 7 8 9]
inv(A)
```

Then start Matlab from the directory containing this file, and enter

```
>> mymatrix
```

The result is the same as if you had entered the two lines of the file, at the prompt.

3.4 Matlab Function

You can write your own functions and add them to the Matlab environment. These functions are just a type of M-file and are created as an ASCII file via a text editor. The first word in the M-file must be the keyword `function` to tell Matlab that this file is to be treated as a function with arguments. On the same line as the word `function` is the calling template that specifies the input and output arguments of the function. The filename for the M-file must end in `.m` and the filename will become the name of the new command for Matlab. For example, consider the following file:

```
function y = foo( x, L )
%FOO get last L points of x
% usage:
%     y = foo( x, L )
% where:
%     x = input vector
%     L = number of points to get
%     y = output vector
N = length(x);
```

```

if( L > N )
    error('input vector too short')
end
y = x((N-L+1):N);

```

If this file is called `foo.m`, the operation could be invoked from the Matlab command by typing

```
aaa = foo(rand(11,1), 7);
```

The effect will be to generate a vector of eleven random numbers and then get the last seven. Note that the variable names inside the function are all local variables and they disappear after the function completes. The argument names `y`, `x`, and `L` are dummy names which are passed values when the function is invoked.

Most functions can be written according to a standard format. Consider a `clip` function M-file that takes two input arguments (a signal vector and a scalar threshold) and returns an output signal vector. You can use an editor to create an ASCII file `clip.m` that contains the following statements:

Eight lines of comments at the beginning of the function will be the response to `help clip`

First step is to figure out matrix dimensions of `x`

Input could be row or column vector

Since `x` is local, we can change it without affecting the workspace

Preserve the sign of `x[n]` create output vector

```

function y = clip( x, Limit )
%CLIP    saturate mag of x[n] at Limit
%        when |x[n]| > Limit, make |x[n]| = Limit
%
% usage: Y = clip( X, Limit )
%
%        X - input signal vector
%        Limit - limiting value
%        Y - output vector after clipping

[nrows ncols] = size(x);

if( ncols ~= 1 & nrows ~= 1 )    %-- NEITHER
    error('CLIP: input not a vector')
end
Lx = max([nrows ncols]);        %-- Length

for n=1:Lx                        %-- Loop over entire vector
    if( abs(x(n)) > Limit )
        x(n) = sign(x(n))*Limit; %-- saturate
    end
end
y = x;                            %-- copy to output vector

```

We can break down the M-file `clip.m` into four elements:

- 1. Definition of Input/Output:** Function M-files must have the word `function` as the very first thing in the file. The information that follows `function` on the same line is a declaration of how the function is to be called and what arguments are to be passed. The name of the function should match the name of the M-file; if there is a conflict, it is the name of the M-file which is known to the Matlab command environment.

Input arguments are listed inside the parentheses following the function name. Each input is a matrix. The output argument (a matrix) is on the left side of the equals sign. Multiple output arguments are also possible, e.g., notice how the function `size(x)` in `clip.m` returns the number of rows and number of columns into separate output variables. Square brackets

surround the list of output arguments. Finally, observe that there is no explicit return of the outputs; instead, Matlab **returns** whatever value is contained in the output matrix when the function completes. The Matlab function return just leaves the function, it does not take an argument. For **clip** the last line of the function assigns the clipped vector to **y**.

The essential difference between the function M-file and the script M-file is dummy variables versus permanent variables. The following statement creates a clipped vector **wclipped** from the input vector **win**.

```
>> wclipped = clip(win, 0.9999);
```

The arrays **win** and **wclipped** are permanent variables in the workspace. The temporary arrays created inside **clip** (i.e., **y**, **nrows**, **ncols**, **Lx** and **i**) exist only while **clip** runs; then they are deleted. Furthermore, these variable names are local to **clip.m** so the name **x** could also be used in the workspace as a permanent name. These ideas should be familiar to anyone with experience using a high-level computer language like C, FORTRAN or PASCAL.

- 2. Self-Documentation:** A line beginning with the % sign is a comment line. The first group of these in a function are used by Matlab's help facility to make M-files automatically self-documenting. That is, you can now type **help clip** and the comment lines from your M-file will appear on the screen as help information!! The format suggested in **clip.m** follows the convention of giving the function name, its calling sequence, definition of arguments and a brief explanation.
- 3. Size and Error Checking:** The function should determine the size of each vector/matrix that it will operate on. This information does not have to be passed as a separate input argument, but can be extracted on the fly with the **size** function. In the case of the **clip** function, we want to restrict the function to operating on vectors, but we would like to permit either a row ($1 \times L$) or a column ($L \times 1$). Therefore, one of the variables **nrows** or **ncols** must be equal to one; if not we terminate the function with the bail out function **error** which prints a message to the command line and quits the function.
- 4. Actual Function Operations:** In the case of the **clip** function, the actual clipping is done by a for loop which examines each element of the **x** vector for its size versus the threshold **Limit**. In the case of negative numbers the clipped value must be set to **-Limit**, hence the multiplication by **sign(x(n))**. This assumes that **Limit** is passed in as a positive number, a fact that might also be tested in the error checking phase.

3.5 Debugging a Matlab M-file

Since Matlab is an interactive environment, debugging can be done by examining variables in the workspace. Matlab versions 4 and 5 contain a debugger with support for break points. Several useful debugging functions include **dbstop**, **dbup**, **dbstep**, **dbcont**, **dbquit**, and **keyboard**.

In addition, Matlab also allows you to stop at specific points in your .m files, examine the workspace and step through execution of your code. We will briefly discuss some of these commands. This first command is

```
>> pause
```

This stops an M-file until any key is pressed. It is very useful after plotting commands; `pause(n)`, will pause for **n** seconds and `pause(-2)` cancels all subsequent pauses.

To displays the program contents while an M-file is being executed, you can use

```
>> echo
or >> echo on
or >> echo off
```

This is a toggle command, so typing `echo` by itself will change the echo state. If **echo** is on, it affects all script files; it affects function files differently, though. To turn echo on for a particular function file, use `echo filename on`; `echo on` all turns it on for all functions.

To stops an M-file and gives the control to the keyboard, you can use

```
keyboard
```

With the `keyboard` command, you can view and change all variables as you wish. Typing return and pressing the ENTER key allows the M-file to continue. It is very useful for programs debugging.

3.6 Programming Tips

This section presents a few programming tips that should help improve your Matlab programs. For more ideas and tips, list some of the functions M-files in the toolboxes of Matlab using the **type** command. Copying the style of other programmers is always an efficient way to improve your own knowledge of a computer language. In the hints below, we discuss some of the most important points involved in writing good Matlab code. These comments assume that you are both an experienced programmer and have already been introduced to Matlab.

Avoiding FOR Loops

Since Matlab is an interpreted language, certain common programming habits are intrinsically inefficient. The primary one is the use of **for** loops to perform simple operations over an entire matrix or vector. Whenever possible, you should try to find a vector function (or the composition of a few vector functions) that will accomplish the same result rather than writing a loop. For example, if the operation were summing up all the elements in a matrix, the difference between calling `sum` and writing a loop that looks like *Fortran* code can be astounding---the loop is unbelievably slow due to the interpretative nature of Matlab. Consider the following three methods for matrix summation:

Double Loop
needed to
index all
matrix entries

```
[Nrows, Ncols] = size(x);
xsum = 0.0;
for m = 1:Nrows
    for n = 1:Ncols
        xsum = xsum + x(m,n);
    end
end
```

sum acts on a matrix
to give the sum down
each column

```
xsum = sum(sum(x));
```

$\mathbf{x}(:)$ is a vector of all elements in the matrix

```
xsum = sum( x(:) );
```

The last two methods rely on the built-in function `sum` which has different characteristics depending on whether its argument is a matrix or a vector (called "operator overloading"). To get the third (and most efficient) method, the matrix \mathbf{x} is converted to a column vector with the colon operator. Then one call to `sum` will suffice.

Repeating Rows or Columns

Often it is necessary to form a matrix by replicating a value in the rows or columns. If the matrix is to have all the same values, then functions such `ones(M,N)` and `zeros(M,N)` can be used. But when you want to replicate a column vector \mathbf{x} to create a matrix that has eleven identical columns, you can avoid a loop by using the outer-product matrix multiply operation. The following Matlab code fragment will do the job:

```
X = x * ones(1,11)
```

If \mathbf{x} is a length L columns vector, then the matrix \mathbf{X} formed by the outer product is $L \times 11$.

Vectorizing Logical Operations

The `clip` function offers a different opportunity for vectorization. The `for loop` in that function contains a logical test and might not seem like a candidate for vector operations. However, the logical operators in Matlab apply to matrices. For example, a greater than test applied to a 3×3 matrix returns a 3×3 matrix of ones and zeros.

```
>> x = [ 1 2 -3; 3 -2 1; 4 0 -1]
>> x = [ 1 2 -3
      3 -2 1
      4 0 -1 ]
>> mxx = x>0                                %-- check the greater than condition
>> mxx = [ 1 1 0
      1 0 1
      1 0 0 ]
>> y = mxx .* x                               %-- multiply by masking matrix
>> y = [ 1 2 0
      3 0 1
      4 0 0 ]
```

The zeros mark where the condition was false; the ones denote true. Thus, when we multiply \mathbf{x} by the masking matrix `mxx`, we get a result that has all negative elements set to zero. Note that the entire matrix has been processed without using a loop. Since the saturation done in `clip.m` requires that we change the large values in \mathbf{x} , we can implement the for loop with three array multiplications. This leads to a vectorized saturation operator that works for matrices as well as vectors:

```
y = x.*(abs(x)<=Limit) + Limit*(x>Limit) - Limit*(x<-Limit);
```

Three different masking matrices are needed to represent the three cases of negative saturation, positive saturation, and no action. The additions correspond to the logical OR of these cases. The number of arithmetic operations needed to carry out this statement is $3N$ multiplications and $2N$ additions where N is the total number of elements in \mathbf{x} . On the other hand, the statement is interpreted only once.

Creating Impulse and Step Sequences

Another simple example is given by the following trick for creating an impulse and step signal vector:

```
nn = [-20:80];
impulse = (nn==0);
step = (nn>=0);
```

This result could be plotted with `stem(nn, impulse)`. In some sense, this code fragment is perfect because it captures the essence of the mathematical formula which defines the impulse and step functions as

$$\delta(n) = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases} \quad u(n) = \begin{cases} 1 & n \geq 0 \\ 0 & n < 0 \end{cases}$$

The Find Function

An alternative to masking is to use the find function. This is not necessarily more efficient; it just gives a different approach. The `find` function will determine the list of indices in a vector where a condition is true. For example, `find(x>Limit)`; will return the list of indices where the vector is greater than the `Limit` value. Thus we can do saturation as follows:

```
y = x;
jkl = find(y>Limit);
y( jkl ) = Limit*ones( size(jkl) );
jkl = find(y<-Limit);
y( jkl ) = -Limit*ones( size(jkl) );
```

The ones function is needed to create a vector on the right-hand side that is the same size as the number of elements in `jkl`. In version 5.0 this would be unnecessary since a scalar assigned to a vector is now assigned to each element of the vector.

Seek to Vectorize

The dictum to "avoid for loops" is not always an easy path to follow, because it means the algorithm must be cast in a vector form. We have seen that this is not particularly easy when the loop contains a logical test, but such loops can still be "vectorized" if masks are created for all possible conditions. This does result in extra arithmetic operations, but they will be done efficiently by the internal vector routines of Matlab, so the final result should still be much faster than an interpreted for loop.

Programming Style

"May your functions be short and your variable names long." Each function should have a single purpose. This will lead to short simple modules that can be composed together with other functions to produce more complex operations. Avoid the temptation to build super functions with many options and a plethora of outputs.

Matlab supports long variable names---up to 32 characters. Take advantage of this feature to give variables descriptive names. In this way, the number of comments littering the code can be

drastically reduced. Comments should be limited to help information and documentation of tricks used in the code.

Continuing Long Lines

A long Matlab command may be broken onto two lines by placing an ellipses (...) at the end of the line to be continued.

4. Reference

- [1] Duane Hanselman and Bruce Littlefield, Mastering MATLAB, Prentice Hall, ISBN 0-13-191594-0, 542 pages, 1996.
- [2] Kermit Signon, Matlab Primer, CRC Press, Boca Raton, FL, ISBN 0-8493-9440-6, 1994.
- [3] The MathWorks, Inc., The Student Edition of MATLAB Version 4 User's Guide, Prentice Hall, ISBN 0-13-184979-4, 1995.
- [4] The MathWorks, Inc., The Student Edition of SIMULINK User's Guide, Prentice Hall, ISBN 0-13-452435-7, 1995.
- [5] C. Sidney Burrus, James H. McClellan, Alan V. Oppenheim, Thomas W. Parks, Ronald W. Schafer, and Hans Schuessler, Computer-Based Exercises for Signal Processing Using MATLAB, Prentice Hall, ISBN 0-13-219825-8, 1994.
- [6] Leland B. Jackson, Digital Filters and Signal Processing, 3e - with MATLAB Exercises, Kluwer Academic Publishers, ISBN 0-7923-9559-X, 1995.