The unessential, hopefully easy and quick-access

reference guide to Matlab

- an intro and a guide to programming in Matlab –

Psych 403/505

Matlab for Vision Research

Information assembled from Matlab’s help desk and Dr. Alejandro Lleras at U Illinois.

Important warning regarding antivirus software:

Antivirus software can be set to “scan” information that you are writing and/or reading to memory (in some programs, this is the default). When running experiments, this means that every time you save a variable or read a variable, your antivirus software gets in the way and checks it for viruses. While a noble attempt at computer safety, this will have serious impact in the performance of Matlab, particularly, on timing-sensitive experiments. My personal recommendation is to turn off your antivirus software and never connect your running computers to the internet (or even to local networks).

Important warning regarding Matlab network licenses:

Matlab network licenses should not be used to run experiments. These versions of Matlab require your computer to be hooked-up to a network, which is most likely linked to the www. Besides the warning above, having your computer connected to the net will significantly impact the timing of your experiments, as your computers deals with network communications (information passing through your computer to reach another terminal in the network). In addition, Matlab itself will communicate with a central machine in the network every 15 minutes or so to verify that your copy of the program is indeed licensed. Not Good and definitely paranoid computer behavior.

TABLE OF CONTENTS

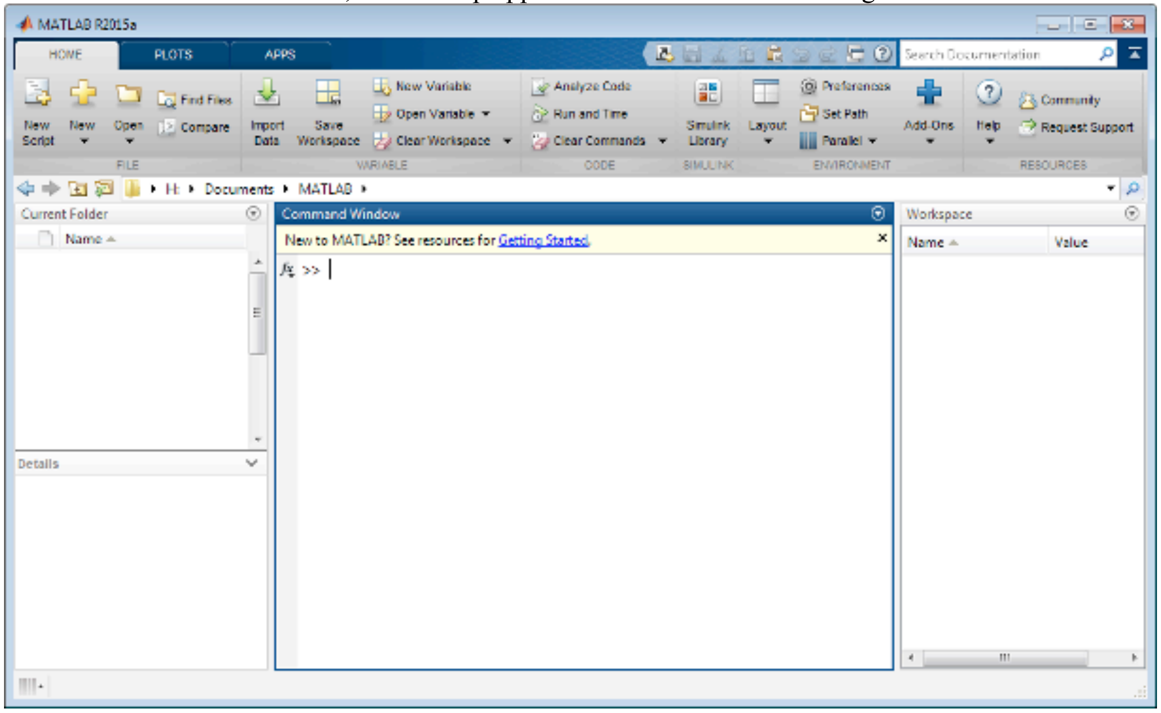
MATLAB introduction and general guide

For general Matlab related questions, visit the website below:
http://www.mathworks.com/access/helpdesk/help/techdoc/

MATLAB Desktop

When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB.

The first time MATLAB starts, the desktop appears as shown in the following illustration.



Command	Description	Example

cd	Display current directory
	Change current directory
		`cd`
		`cd c:\experiments\hoo`

dir	Display file names in current directory	`dir`
edit	Launch MATLAB editor	`edit hoo.m`
delete	Delete files	`delete hoo.m`
help	Display help for MATLAB functions in Command Window	`help Screen`
helpdesk	Display Help browser	`helpdesk`

For running experiments, it might be safer to use MATLAB with -nojvm option, because the window system (above) does not always accept the `getchar` function.

Variables

Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB uses only the first 31 characters of a variable name. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. A and a are not the same variable. To view the matrix assigned to any variable, simply enter the variable name. To assign a value to a variable, simply type its name, followed by the equal sign followed by the value you want to assign. For example, the variable myage will be assigned the value 29 by typing:
myage = 29
For letter strings, surround the value by quote marks to distinguish them from variable names. For example, the variable myname will be assigned the value alejo by typing:
myname = 'alejo'
Typing:
myname = alejo
treats alejo as a variable (not as a value) and transfers the value of the variable alejo to the variable myname.

- You do not need to type or declare variables used in M-files, (with the possible exception of designating them as global or persistent).
- Before assigning one variable to another, you must be sure that the variable on the right-hand side of the assignment has a value. In other words, B = A works only if A has a value.
- Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.

- Have your own CONSISTENT rules for Variable Names
- Make Sure Variable Names Are Valid
- Don't Use Function Names for Variables
- Check for Reserved Keywords
- Avoid Using i and j for Variables
- Avoid Overwriting Variables in Scripts

Global Variables: If you want more than one function to share a single copy of a variable, simply declare the variable as global in all the functions. Do the same thing at the command line if you want the base workspace to access the variable. The global declaration must occur before the variable is actually used in a function. Although it is not required, using capital letters for the names of global variables helps distinguish them from other variables. For example, create an M-file called `falling.m`.

```
➢  function h = falling(t)
➢  global GRAVITY
➢  h = 1/2*GRAVITY*t.^2;
```

Persistent Variables: If you want the same function to keep using the same variable, on subsequent runs of the function, use a persistent variable.

Characteristics of persistent variables are

- You can declare and use them within M-file functions only.
- Only the function in which the variables are declared is allowed access to it.
- MATLAB does not clear them from memory when the function exits, so their value is retained from one function call to the next.

You must declare persistent variables before you can use them in a function. It is usually best to put your persistent declarations toward the beginning of the function. You would declare persistent variable `SUM_X` as follows:

```
•    persistent SUM_X
```

If you clear a function that defines a persistent variable (i.e., using `clear` *`functionname`* or `clear all`), or if you edit the M-file for that function, MATLAB clears all persistent variables used in that function.

You can use the `mlock` function to keep an M-file from being cleared from memory, thus keeping persistent variables in the M-file from being cleared as well.

Initializing Persistent Variables. When you declare a persistent variable, MATLAB initializes its value to an empty matrix, `[]`. After the declaration statement, you can assign your own value to it. This is often done using an `isempty` statement, as shown here:

```
•    function findSum(inputvalue)
•    persistent SUM_X
•
•    if isempty(SUM_X)
•       SUM_X = 0;
•    end
•    SUM_X = SUM_X + inputvalue
```

This initializes the variable to 0 the first time you execute the function, and then accumulates the value on each iteration.

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Right division
\	Left division
^	Power
:	Colon operator

(continued…)
The colon, :, is one of the most important MATLAB operators. It occurs in several different forms. The expression

```
1:10
```

is a row vector containing the integers from 1 to 10:

```
1    2    3    4    5    6    7    8    9    10
```

To obtain nonunit spacing, specify an increment. For example,

```
100:-7:50
```

is
100 93 86 79 72 65 58 51

Subscript expressions involving colons refer to portions of a matrix:

    A(1:k,j)

is the first k elements of the jth column of A. So:

    sum(A(1:4,6))

computes the sum of the first four elements of the sixth column. But there are other ways. The colon by itself refers to all the elements in a row or column of a matrix and the keyword end refers to the last row or column. So:

    sum(A(:,end))

computes the sum of the elements in the last column of A

Relational Operators

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Logical Operators

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
E = [1 2
     5 3];
```

Operator	Description	Example
&	Returns 1 for every element location that is true (nonzero) in both arrays, and 0 for all other elements.	A & B = 01001
|	Returns 1 for every element location that is true (nonzero) in either one or the other, or both, arrays and 0 for all other elements.	A | B = 11101
~	Complements each element of input array, A.	~A = 10010
xor	Returns 1 for every element location that is true (nonzero) in only one array, and 0 for all other elements.	xor(A,B)=10100

Matlab Common Symbols and Operations

Symbol	Description	Example
*	An asterisk in a filename specification is used as a wildcard specifier.	Dir('January_*.mat) Will find all m files containing January_*.
:	The colon operator generates a sequence of numbers that you can use in creating or indexing into arrays.	--C = A(1, 2: 4) Creates C = [1 1 0] -- N = 1:40 Creates a sequence of incremental numbers from 1 to 40. You can also use the syntax -N= first:step:last For other increments
,	A comma is used to separate the following types of elements.	-Separate elements belonging in the same row C=[1,1,0] -Separate indexes of an array D=E(1,2) D=[2] -Separate arguments when calling a function.
.	Add fields to a MATLAB structure by following the structure name with a dot and then a field name:	-DATA(i).rt = reactiontime -DATA(i).trial = i
…	To continue a Matlab statement on a different line	
( )	Parentheses are used mostly for indexing into elements of an array or for specifying arguments passed to a called function.	A(1)
;	The semicolon can be used to construct arrays, suppress output from a MATLAB command, or to separate commands entered on the same line.	`E = [1 2;5 3]`
' '	Single quotes are the constructor symbol for MATLAB character arrays.	`S = 'Hello World'`
[ ]	Square brackets are used to delimitate the contents of a matrix. But empty square brackets are used to DELETE rows or columns of a matrix.	`E(:,2)=[]` `E = [1` `5]`

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1. Parentheses ()
2. Transpose (.'), power (.^), complex conjugate transpose ('), matrix power (^)
3. Unary plus (+), unary minus (-), logical negation (~)
4. Multiplication (.*), right division (./), left division(.\), matrix multiplication (*), matrix right division (/), matrix left division (\)
5. Addition (+), subtraction (-)
6. Colon operator (:)
7. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
8. Element-wise AND (&)
9. Element-wise OR (|)
10. Short-circuit AND (&&) (second operation not evaluated if first one is false)
11. Short-circuit OR (||) (second operation is not evaluated if first one is true)

Some mathematical functions

`Y = round(X)`	Round to nearest integer
`Y = fix(X)`	Round towards zero
`Y = ceil(X)`	Round toward infinity
`Y = floor(X)`	Round towards minus infinity
`Y = mod(N,n)`	Modulus after division
`Y = rem(N,n)`	Remainder after division
`Y = abs(X)`	Absolute value and complex magnitude
`Y = sqrt(X)`	Square root

Keywords

MATLAB reserves certain words for its own use as keywords of the language. To list the keywords, type

```
    iskeyword
    ans =
'break'                                  'function'
'case'                                   'global'
'catch'                                  'if'
‘classdef’                               'otherwise'
'continue'                               'persistent'
'else'                                   'return'
'elseif'                                 'switch'
'end'                                    'try'
'for'                                    'while'
```

An important note on memory allocation

Matlab does not require that you say what precision (= how much memory space) to use to store your variables. The default for numbers is "double", meaning a double-precision floating point, which can store a variable with great precision (uses 64 bits, 8 bytes). Range goes from:
-1000000000000000000000000000000…..0000000000000000000000000000000 (with 308 "zeros"), which is pretty close to an infinite negative number to:
-0.0000000….00000002 (with 308 "zeros"), which is pretty close to zero, on the negative side.

Plus the same precision on positive numbers.

I hope you will agree that most often you WON'T need that much precision.

Other, smaller formats are:

- Int8 and Uint8 (2^8 = 256 total integer values). The U stands for Unsigned, meaning only positive numbers are stored. Zero is as low as you can go.
- Int16 and Uint16 (2^16 = 65536 total integer values).
- There are also, Int32, Uint32, Int64 and Uint64.

BUT BEWARE: magnitudes are CUT-OFF at the limits of a variable class.

For real numbers (requiring periods and such) you can also use single-precision floating points ("single"), which only use 32 bits of storage.

Related functions are below, to help you change the format of numbers.
double Convert to double-precision
int8 Convert to signed 8-bit integer
int16 Convert to signed 16-bit integer
int32 Convert to signed 32-bit integer
int64 Convert to signed 64-bit integer
single Convert to single-precision
uint8 Convert to unsigned 8-bit integer
uint16 Convert to unsigned 16-bit integer
uint32 Convert to unsigned 32-bit integer
uint64 Convert to unsigned 64-bit integer

Because numbers are also characters, you can choose to store them as such or revert characters into numbers as well, using:

str2double Convert string to double-precision number
str2num Convert string to number
int2str Convert integer to string
num2str Convert number to string
char Convert to character array (string)

Programming:

Flow Control

There are eight flow control statements in MATLAB:

- `if`, together with else and `elseif`, executes a group of statements based on some logical condition.
- `switch`, together with `case` and `otherwise`, executes different groups of statements depending on the value of some logical condition.
- `while` executes a group of statements an indefinite number of times, based on some logical condition.
- `for` executes a group of statements a fixed number of times.
- `continue` passes control to the next iteration of a `for` or `while` loop, skipping any remaining statements in the body of the loop.
- `break` terminates execution of a `for` or `while` loop.
- `try`...`catch` changes flow control if an error is detected during execution.
- `return` causes execution to return to the invoking function.

All flow constructs use `end` to indicate the end of the flow control block.

if	for	while	switch
`if ...`	`for n=1:10`	`while`	`switch n`
`...`	`...`	`...`	`case n1`
`[elseif`	`end`	`if ...`	`...`
`...`		`break`	`case n2`
`else`		`end`	`...`
`...`		`end`	`otherwise`
`end]`			`...`
`end`			`end`

Note on Speed

Preallocation You can make your `for` loops go faster by preallocating any vectors or arrays in which output results are stored. For example, this code uses the function `zeros` to preallocate the vector created in the `for` loop. This makes the `for` loop execute significantly faster.

```
•    r = zeros(32,1);
•    for n = 1:32
•        r(n) = rank(magic(n));
•    End
```

Without the preallocation in the previous example, the MATLAB interpreter enlarges the `r` vector by one element each time through the loop. This may require finding a new location in memory to fit the newly resized variable. All of this takes time. Vector preallocation eliminates this and results in faster execution.

Structure

Structures are multidimensional MATLAB arrays with elements accessed by textual field designators. For example,

```
D.trial = 1;
D.setsize = 3;
D.tgt_id = 2;
D.rt = 541;
D.resp = 1;
```

creates a structure with five fields.

To view this structure, on the command line type

D

You can make this structure as an array. The following example creates 100 elements of each field. To create an array of structure, you need dummy values. The number after each field definition is the dummy which can be replaced with other values somewhere in the program.

```
DATA=repmat(struct('trial',1,'setsize',1,'tid',1,'rt',1,'resp',1
),100,1);
```

To access this structure, for example to know the setsize at Trial 21, do like this.

```
DATA(21).setsize
```

You also make arrays in structures. For example, when you need two targets, as in AB experiments, define the structure like this.

```
DATA(64)=struct('trial',1,'lag',1,'tid',[1,2],'resp',[1,2]);
```

What is an m-file?

An m-file is a text file containing MATLAB code. Use the MATLAB Editor or another text editor to create a file containing the same statements you would type at the MATLAB command line. Save the file under a name that ends in .m.

For example, create a file containing these five lines:

```
A = [ ...
16.0   3.0   2.0  13.0
 5.0  10.0  11.0   8.0
 9.0   6.0   7.0  12.0
 4.0  15.0  14.0   1.0 ];
```

Store the file under the name magik.m. Then the statement

magik

reads the file and creates a variable, A, containing our example matrix.

Of course, the m-files we will be creating are more complex than that. But in general, m-files are simply a list of command-line instructions that will be **executed in the order in which they are read** (by the Matlab interpreter).

There are two kinds of M-files:

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace. This is what we will refer to as a "Matlab program".
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. In addition, scripts can produce graphical output using functions like plot.

NOTE ON MEMORY USE/LIMITATIONS: Because scripts (and therefore the programs we will be creating) leave the variables they created during their execution in the computer's memory (matlab's workspace), if you are not diligent about erasing unneeded information, you can quickly fill-up your workspace memory or delay the speed at which your RAM is read/written to. This can significantly impact the timing in your experiments.

Functions are M-files that can accept input arguments and return output arguments. The names of the M-file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.
The input argument is also called a PASSING VARIABLE, because you are passing a variable to be used by the function you are calling.

Other useful stuff

To randomize, say

```
rand('state',sum(100*clock));
```

at the beginning of the program. Then, if you need a random number from 0-9, use this.

```
tempx= fix(10*rand(1));
```

If you need 1-10,

```
tempx= fix(10*rand(1))+1;
```

If you want to have numbers 1-100 in a random order, do this:

```
seq=randperm(100);
```

The array `seq` has 100 elements with random order.

You can also use **shuffle:**
[Y, index] = shuffle(X)

which means that Y=X(index).

Or simply,
Y = shuffle(X)
if you don't care to keep track of the old indexes.

Linspace may also be useful.

```
x=linspace(0,10,6)            x= 0 2 4 6 8 10
```

Other commands worth knowing:

`hidecursor`	Hide cursor
`showcursor`	Show cursor
`Clc`	Clear command window
`Clear X`	Clear variable X
`Clear all`	Clear all variables (including MATLAB window)

MATLAB provides four functions that generate basic matrices.

zeros	All zeros
ones	All ones
rand	Uniformly distributed random elements
randn	Normally distributed random elements

```
Examples:

    Z = zeros(2,4)
    Z =
         0      0      0      0
         0      0      0      0


    F = 5*ones(3,3)
    F =
         5      5      5
         5      5      5
         5      5      5
```

```
N = fix(10*rand(1,10))
N =
     9     2     6     4     8     7     4     0     8     4

R = randn(4,4)
R =
    0.6353    0.0860   -0.3210   -1.2316
   -0.6014   -2.0046    1.2366    1.0556
    0.5512   -0.4931   -0.6313   -0.1132
   -1.0998    0.4620   -2.3252    0.3792
```

sprintf function

Write formatted data to a string

Syntax

```
[s,errmsg] = sprintf(format,A,...)
```

Description
`[s,errmsg] = sprintf(format,A,...)` formats the data in matrix A (and in any additional matrix arguments) under control of the specified format string, and returns it in the MATLAB string variable `s`. The `sprintf` function returns an error message string errmsg if an error occurred. `errmsg` is an empty matrix if no error occurred.

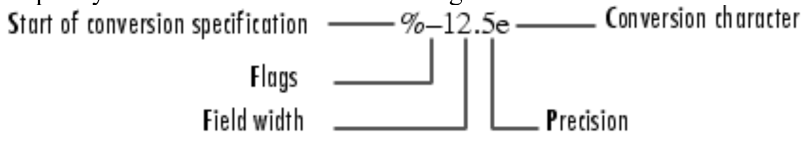
`sprintf` is the same as `fprintf` except that it returns the data in a MATLAB string variable rather than writing it to a file.

Format String
The format argument is a string containing C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent non-printing characters such as newline characters and tabs.

Conversion specifications begin with the % character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:



Flags are explained in the next page.

Flags

You can control the alignment of the output using any of these optional flags.

Character	Description	Example
A minus sign (-)	Left-justifies the converted argument in its field.	`%-5.2d`
A plus sign (+)	Always prints a sign character (+ or -).	`%+5.2d`
Zero (0)	Pad with zeros rather than spaces.	`%05.2d`

Field Width and Precision Specifications

You can control the width and precision of the output by including these options in the format string.

Character	Description	Example
Field width	A digit string specifying the **minimum** number of digits to be printed.	`%6f`
Precision	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.	%6.2f

Conversion Characters

Conversion characters specify the notation of the output.

Specifier	Description
%c	Single character
%d	Decimal notation (signed)
%e	Exponential notation (using a lowercase e as in 3.1415e+00)
%E	Exponential notation (using an uppercase E as in 3.1415E+00)
%f	Fixed-point notation
%g	The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.
%G	Same as %g, but using an uppercase E
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)
%x	Hexadecimal notation (using lowercase letters a-f)
%X	Hexadecimal notation (using uppercase letters A-F)

Escape Characters

This table lists the escape character sequences you use to specify non-printing characters in a format specification.

Character	Description
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\‘ ‘ or ‘ ‘ (two single quotes)	Single quotation mark
%%	Percent character

The eval Function

The eval function works with text variables to implement a powerful text macro facility. The expression or statement

```
•   eval(s)
```

uses the MATLAB interpreter to evaluate the expression or execute the statement contained in the text string `s`.

For example

```
•   for d = 1:31
•      s = ['load Subject' int2str(d) '.dat'];
•      eval(s)
•      % Loads one by one the contents of the d-th Subject file
•      % Subject1.dat then Subject2.dat, and Subject3.dat, etc
•   end
```

The repmat Function

This is another useful function in Matlab that allows you to replicate and tile matrices.

- B = repmat(A,M,N) creates a large matrix B consisting of an M-by-N tiling of copies of A. see also the concat function, which allows you to tile to different matrices together.

Reading Strings Line by Line from Text Files

MATLAB provides two functions, `fgetl` and `fgets`, that read lines from formatted text files and store them in string vectors. The two functions are almost identical; the only difference is that `fgets` copies the newline character to the string vector but `fgetl` does not.

The following M-file function demonstrates a possible use of `fgetl`. This function uses `fgetl` to read an entire file one line at a time. For each line, the function determines whether an input literal string (`literal`) appears in the line.

If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
• function y = litcount(filename, literal)  %filename and literal
                                             are passing variables
• % Search for number of string matches per line.
• % filename is the file you’re looking through
• % Literal is the string your looking for
•
```

```
•  fid = fopen(filename, 'rt');    %f + open, opens the file
•                                   % to read = r
•                                   % as text = t
•  y = 0;
•  while feof(fid) == 0    %until the End Of the File= eof
•     tline = fgetl(fid);  %get line= getl
•     matches = findstr(tline, literal);
•     num = length(matches);
•     if num > 0
•        y = y + num;
•        fprintf(1,'%d:%s\n',num,tline);
•     end
•  end
•  fclose(fid);
```

Reading Formatted ASCII Data

The `fscanf` function is like the `fscanf` function in standard C. Both functions operate in a similar manner, reading data from a file and assigning it to one or more variables. Both functions use the same set of conversion specifiers to control the interpretation of the input data.

The conversion specifiers for `fscanf` begin with a `%` character; common conversion specifiers include

Conversion Specifier	Description
`%s`	Match a string.
`%d`	Match an integer in base 10 format.
`%g`	Match a double-precision floating-point value.

You can also specify that `fscanf` skip a value by specifying an asterisk in a conversion specifier. For example, `%*f` means skip the floating-point value in the input data; `%*d` means skip the integer value in the input data.

Differences Between the MATLAB fscanf and the C fscanf

Despite all the similarities between the MATLAB and C versions of `fscanf`, there are some significant differences. For example, consider a file named `moon.dat` for which the contents are as follows:

```
•    3.654234533
•    2.71343142314
•    5.34134135678
```

The following code reads all three elements of this file into a matrix named `MyData`:

- `fid = fopen('moon.dat','r');`
- `MyData = fscanf(fid,'%g');`
- `status = fclose(fid);`

Notice that this code does not use any loops. Instead, the `fscanf` function continues to read in text as long as the input format is compatible with the format specifier.

An optional size argument controls the number of matrix elements read. For example, if `fid` refers to an open file containing strings of integers, then this line reads 100 integer values into the column vector `A`:

- `A = fscanf(fid,'%5d',100);`

This line reads 100 integer values into the 10-by-10 matrix `A`:

- `A = fscanf(fid,'%5d',[10 10]);`

A related function, `sscanf`, takes its input from a string instead of a file. For example, this line returns a column vector containing `2` and its square root:

- `root2 = num2str([2, sqrt(2)]);`
- `rootvalues = sscanf(root2,'%f');`

Writing Formatted Text Files

The fprintf function converts data to character strings and outputs them to the Screen or a file. A format control string containing conversion specifiers and any optional text specify the output format. The conversion specifiers control the output of array elements; `fprintf` copies text directly.

Common conversion specifiers include

Conversion Specifier	**Description**
`%e`	Exponential notation
`%f`	Fixed-point notation
`%g`	Automatically select the shorter of `%e` and `%f`

Optional fields in the format specifier control the minimum field width and precision. For example, this code creates a text file containing a short table of the exponential function:

- `x = 0:0.1:1;`
- `y = [x; exp(x)];`

The code below writes `x` and `y` into a newly created file named `exptable.txt`:

- `fid = fopen('exptable.txt','w');`
- `fprintf(fid,'Exponential Function\n\n');`
- `fprintf(fid,'%6.2f  %12.8f\n',y);`
- `status = fclose(fid);`

The first call to `fprintf` outputs a title, followed by two carriage returns. The second call to `fprintf` outputs the table of numbers. The format control string specifies the format for each line of the table:

- A fixed-point value of six characters with two decimal places
- Two spaces
- A fixed-point value of twelve characters with eight decimal places

`fprintf` converts the elements of array `y` in column order. The function uses the format string repeatedly until it converts all the array elements.

Now use `fscanf` to read the exponential data file:

- `fid = fopen('exptable.txt','r');`
- `title = fgetl(fid);`
- `[table,count] = fscanf(fid,'%f %f',[2 11]);`
- `table = table';`
- `status = fclose(fid);`

The second line reads the file title. The third line reads the table of values, two floating-point values on each line, until it reaches end of file. `count` returns the number of values matched.

A function related to `fprintf`, `sprintf`, outputs its results to a string instead of a file or the Screen. For example,

- `root2 = sprintf('The square root of %f is %10.8e.\n',2,sqrt(2));`

Help Text

You can create online help for your M-files by entering help text on one or more consecutive comment lines at the start of your M-file program. MATLAB considers the first group of consecutive lines immediately following the H1 line that begin with % to be the online help text for the function. The first line without % as the left-most character ends the help.
The help text for the average function is
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.

When you type `help` *`functionname`* at the command prompt, MATLAB displays the H1 line followed by the online help text for that function. The help system ignores any comment lines that appear after this help block.

Images

The following is taken from Matlab's website:

Two-dimensional arrays can be displayed as *images*, where the array elements determine brightness or color of the images. For example, the statements

- load durer
- whos
- Name   Size   Bytes   Class
-
- X   648x509   2638656   double array
- caption   2x28   112   char array
- map   128x3   3072   double array

load the file durer.mat, adding three variables to the workspace. The matrix X is a 648-by-509 matrix and map is a 128-by-3 matrix that is the **colormap** for this image.
MAT-files, such as durer.mat, are binary files that can be created on one platform and later read by MATLAB on a different platform.

The elements of X are integers between 1 and 128, which serve as indices into the colormap, map. Then

- image(X)
- colormap(map)
- axis image

reproduces Dürer's etching shown in Matlab's website.

The question is WHAT IS A COLORMAP?

Colormap

As you probably are aware of, each color that you see in a computer monitor is created by a combination of three colors (three "guns" on CRT monitors) of varying intensity. These base colors were designed to mimic the way humans perceive colors: the presence of blue, red and green sensitive cones inspired the RGB gun system (R=red, G=green, B=blue) of TVs and monitors.

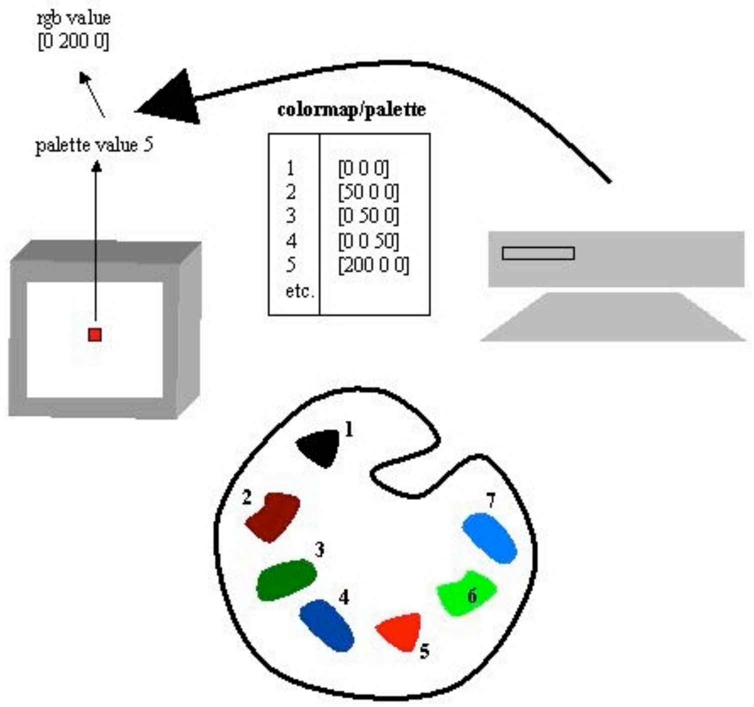
Definition: "A colormap consists of a set of entries defining color values. The colormap associated with a window is used to display the contents of the window; each pixel value indexes the colormap to produce an RGB value that drives the guns of a monitor. Depending on hardware limitations, one or more colormaps can be installed at one time so that windows associated with those maps display with true colors."

In other words, colormaps can be understood as "shortcuts" to your favorite colors (you

can define your own colormaps). In reality, they arose from hardware limitations when it was only possible to manage a small number of colors at a time (16, 32, 256). Today computers can manage an almost “infinite” number of colors. Most monitors now use 8 bits of information to define the intensity of each gun for each pixel in the Screen (known as 24-bit color). That makes:
2^8= 256 levels of red, blue and green.
Meaning
256^3 possible combinations of the three guns = 16.7 million colors (almost infinite).
Newer models have as many as 12 bits per gun (36-bit color) → 68 billions of colors. Hummm….

But most of the time, we don’t need that many. All GIF images, for example, (a widely-used format for images on the web) use only 256 colors. However, which 256 colors are used for a given image depends on the colormap for the image (which is part of the information contained in the GIF file). So different GIFs will have different CLUTs = Color Look-Up Tables, to match CLUT indexes with specific RGB values for the guns.

You can change the appearance of a pixel in an image either by changing the color (RGB value) for that pixel on the image file or by changing the colormap used to read that image.



Two ways of changing color on the screen
1) Change the palette value of that pixel
2) Change the rgb triplet referenced by that palette value.

For example, with the image of Dürer still on your workspace, type:

- image(X)
- colormap(hot)
- axis image

The function hot generates a colormap containing shades of reds, oranges, and yellows. See the colormap reference page for a list of other predefined colormaps.

Reading and Writing Images

You can read standard image files (TIFF, JPEG, BMP, etc.) into MATLAB using the imread function. The type of data returned by imread depends on the type of image you are reading.
You can write MATLAB data to a variety of standard image formats using the imwrite function. See the reference pages for these functions for more information and examples.

SOUNDS

Matlab can play sounds with various commands.

SOUND: sends signal directly to speakers. (soundsc is a related function). But it occupies MATLAB until the end of the sound. The improved Snd command allows Matlab to **do other stuff while the sound is playing**.

WAVPLAY: uses Windows WAVE audio devices and can play sounds **while doing other stuff** (presenting pictures, gathering responses, etc). (see also wavrecord, but don't forget your microphone).

SOUND Play vector as sound.
SOUND(Y,FS) sends the signal in vector Y (with sample frequency FS) out to the speaker on platforms that support sound. Values in Y are assumed to be in the range -1.0 <= y <= 1.0. Values outside that range are clipped (meaning ignored). Stereo sounds are played, on platforms that support it, when Y is an N-by-2 matrix.

SOUND(Y) plays the sound at the default sample rate of 8192 Hz.

Try the following:

- load handel
- sound(y,Fs)

then change the sample frequency (for example).

- Fsslow = Fs/2
- Sound(y,Fsslow)

Snd is an improved version of Sound which gives control back to Matlab while the sound is playing. To achieve this you need to "Open" the sound channel

```
Snd('Open');
```

Then you can "play" a beep:

```
Snd('Play',beep);
```

You can force Matlab to wait while it plays the sound:

```
Snd('Wait');
```

At the end, you should "close" the channel with

```
Snd('Close');    %this also stops currently played sounds.
```

You might want to stop a sound while keeping the sound channel open. Use:

```
Snd('Quiet').
```

MakeBeep(frequency, duration, [samplingRate]) allows you to create a given beep, which you can then play using Snd:

```
beep = MakeBeep(freq,duration);
        Snd('Open');
        .... do some stuff ....
        Snd('Play',beep);
        … do some other stuff while beep plays…
```

WAVPLAY Play sound using Windows audio output device.

WAVPLAY(Y,FS) sends the signal in vector Y with sample frequency of FS Hertz to the Windows WAVE audio device. Standard audio rates are 8000, 11025, 22050, and 44100 Hz. WAVPLAY(Y) automatically sets the sample rate to 11025 Hz. For stereo playback, Y should be an N-by-2 matrix.

> If you want MATLAB to do OTHER STUFF while playing your audio, used WAVPLAY(...,'async'), which begins sound playback and returns immediately from the function call (i.e., a nonblocking call). WAVPLAY(...,'sync') does not return from the function call, until the sound has finished playing (i.e., a blocking call). This is the default playback mode.

Y must contain audio samples stored in double, int16, or uint8 matrices. Double precision data samples must be in the range:
-1.0 <= y <= 1.0; values outside that range are clipped.

You will need to IMPORT the WAV files to your workspace before you can play them (FILE → IMPORT DATA). See also **wavread**

Supported data types for Y and the corresponding number of bits per sample used during playback in each format are as follows:

Data Type	bits/sample
'double'	16
'single'	16
'int16'	16
'uint8'	8

This function is only for use with 32-bit Windows machines.

WAVRECORD Record sound using Windows audio input device (meaning you have to have a microphone installed on your computer!).

WAVRECORD(N,FS,CH) records N audio samples at FS Hertz from CH number of input channels from the Windows WAVE audio device. Standard audio rates are 8000, 11025, 22050, and 44100 Hz. CH can be 1 or 2 (mono or stereo). Samples are returned in a matrix of size N x CH. If not specified, FS=11025 Hz (pretty fast), and CH=1 (mono).

WAVRECORD(..., DTYPE) records and returns data using the data type specified by DTYPE. Supported data types and the corresponding number of bits per sample recorded in each format are as follows:

DTYPE	bits/sample
'double'	16
'single'	16
'int16'	16
'uint8'	8

This function is only for use with 32-bit Windows machines.

Example: Record and play back 5 seconds of 16-bit audio sampled at 11.025 kHz:

```
>       Fs = 11025;
 >      y  = wavrecord(5*Fs, Fs, 'int16');
  >     wavplay(y, Fs);
```

However, the Psychophysics Toolbox community has developed a Audio toolbox that has high onset/offset and playtime precision. It also allows for Matlab to resume control over other stuff while the sound is playing, but this toolbox requires you download a special driver to your computer.

PsychPortAudio

Initialize

InitializePsychSound   is the function that you call to initialize the sound driver.

Open

audiohandle = PsychPortAudio(‘Open’, [ ] , [ ], 0, freq, nrchannels);

This command “opens” the communication to the sound driver on a given sound channel. It is like when we use fopen to start handling text files. audiohandle is the name of the variable (like fid the handle we use to identify the channel we are using). This commands opens the default audio device (first set of [ ]). The second set of brackets indicates the default mode [ ] of only playing playback (not recording). 0 indicates the required latencyclass (low-latency mode). Freq is the frequency that is used for the playback of the audio files and nrchannels the number of channels (mono is 1, stereo is 2).

You can also **record** using PsychPortAudio. To do so, you must open the sound channel on mode 2 (audio capture, instead of playback mode). You also need to specify ahead of time the frequency at which you will be recording. So the Open command would be:
freq = 44100;
pahandle = PsychPortAudio('Open', [], 2, 0, freq, 2);

FillBuffer

PsychPortAudio('FillBuffer', audiohandle, wavedata);

This command fills the playback buffer with the audio inside the file wavedata. It's like when we make a texture in Psychtoolbox before presenting it to the Screen. This is done ahead of the critical moment when you want the audio to play.

Start

t1 = PsychPortAudio('Start', audiohandle, repetitions, 0, 1);

Plays the sound in the buffer indexed by audiohandle for a number of repetitions. 0 means start it immediately (or you can add a delay). 1 means wait for the playback to start and return the timestamp of when that happened, which is saved in the variable t1.

GetStatus

To check if a sound is still being played, get the status of the sound channel with:
s = PsychPortAudio('GetStatus', pahandle);

Stop

PsychPortAudio('Stop', pahandle);   %self explanatory

Close

% Close the audio device: This is very important if you don't want to sacrifice performance of Matlab.
% However, you only need to do it once you are sure you won't be using the same sound channel %again.
PsychPortAudio('Close', pahandle);

GetAudioData

PsychPortAudio('GetAudioData', pahandle, 10);
% Preallocates an internal audio recording variable with a capacity of 10 seconds

When recording, the command 'Start' describes the moment when recording begins when the sound channel is in record mode. This starts the recording until it is manually stopped. 0 is the number of repetitions in this case.
PsychPortAudio('Start', pahandle, 0, 0, 1);

Example:

Voice Trigger:

```
if voicetrigger > 0
    % Yes. Fetch audio data and check against threshold:

  level = 0;

    % Repeat as long as below trigger-threshold:

    while level < voicetrigger

    % Fetch current audiodata:
        [audiodata offset overflow tCaptureStart] = PsychPortAudio('GetAudioData', pahandle);

  % Compute maximum signal amplitude in this chunk of data:
        if ~isempty(audiodata)
            level = max(abs(audiodata(1,:)));
        else
            level = 0;
        end

% Below trigger-threshold?
        if level < voicetrigger

 % Wait for a millisecond before next scan:
            WaitSecs(0.0001);
        end
    end


% Ok, last fetched chunk was above threshold!
    % Find exact location of first above threshold sample.

 idx = min(find(abs(audiodata(1,:)) >= voicetrigger));

    % Initialize our recordedaudio vector with captured data starting from
    % triggersample:
    recordedaudio = audiodata(:, idx:end);

    % For the fun of it, calculate signal onset time in the GetSecs time:
    % Caution: For accurate and reliable results, you should
    % PsychPortAudio('Open',...); the device in low-latency mode, as
    % opposed to the "normal" mode used in this demo! If you fail to do so,
    % the tCaptureStart timestamp may be inaccurate on some systems, and
    % therefore this tOnset timestamp may be off! See for example
    % PsychPortAudioTimingTest and AudioFeedbackLatencyTest for how to
    % setup low-latency high precision mode.
    tOnset = tCaptureStart + ((offset + idx - 1) / freq);

    fprintf('Estimated signal onset time is %f secs, this is %f msecs after start of capture.\n', tOnset,
(tOnset - tCaptureStart)*1000);
```

```
else
    % Start with empty sound vector:
    recordedaudio = [];
end
```

This code will check every millisecond to see if sound has been recorded by the microphone. If so, it begins recording.

Psychophysics Toolbox Screen function

Psychtoolbox-3 still runs under Microsoft Windows XP, Windows Vista, Windows 7 and Windows 8, but we don't test for compatibility with any system but Windows 7, and won't provide any bug fixes or troubleshooting help for any issues that can't be shown to be also present on Windows 7. Specifically, moving away from Windows XP and Vista is strongly recommended. We do aim to keep the toolbox working under these and future versions of Windows, but full support for all features is a lower priority for us than Linux. As of 1. December 2009, Windows Vista and Windows-7 have been tested for basic compatibility with PTB-3. Precision of sound presentation hasn't been tested at all due to lack of suitable testing equipment. Test of visual stimulus presentation on 3 test setups showed somewhat mixed results, especially dual display presentation and presentation timing were rather disappointing. We are aware that over 1300 people do run the toolbox under Vista or Windows-7 and we didn't receive many reports of trouble so far, but we can't recommend it at all for dual-display stereo stimulus presentation or for tasks with a need for high visual timing precision. For some caveats wrt. Vista and later see our FAQ entry about Vista and Windows-7. All in all you are off worse with Vista or Windows-7 instead of XP with Psychtoolbox, so there is no reason to switch to it. Vista et al. seem to provide less performance than XP while at the same time posing higher hardware requirements.

Attention: PsychToolbox-3 runs best in 64bit Matlab (tested with release 2012a)

Below are the most often used functions of the SCREEN function. For a more complete guide on these functions, add a question mark to the function attribute you wish to learn more about: Screen('nameoffunction?').

OpenWindow

[windowPtr,rect]=Screen('**OpenWindow**',windowPtrOrScreenNumber[,color][,rect][,pixelSize]);

windowPtr is a double that will help identify the window (or type of Screen) that we'll be working on. You can call it whatever you like. I usually call it window.

Rect (if specified, and I suggest you do) gives you the coordinates of the window you'll be using in pixels, on the format [Xtop-left, Ytop-left, Xbotton-right, Ybottom-right].

REMEMBER: the origin (point 0,0 for drawing) is at the top left of the Screen. This is the easiest way for Matlab to turn numbers into pictures.

WindowptrorScreenNumber: is the Screen number for the main Screen (where we'll be presenting stimuli).

color: specifies what color you'd like the Screen to be. You can use a scalar (a clut index and therefore a number between 0 and 255) or an RGB value in the format [r g b], in which r, g and b are values between 0 and 255.

rect: inside the Screen command it specifies the size you want the Screen to be. In windows, this value does nothing and the default is the full Screen.

PixelSize: sets the depth (in bits) of each pixel; default is to leave depth unchanged.

Opening / closing a window takes about a second. So you want to do that AHEAD of time, not during your trials.

Note: If you have multiple monitors, you can specify which monitor to use, which is nice since you can have a program running on one Screen while debugging on the other. Screen(‘Screens’) tells you all the available ones. For example, if you have two monitors, Screen(‘Screens’) returns 0 1 2; if you have only one monitor, Screen(‘Screens’) returns 0. It's different on Mac and PC. Suppose you have two monitors, when you type Screen('OpenWindow',**0**,255) or something like that, the 0 refers to a monitor on a Mac, that's just one of the two monitors (the primary one), but on a PC, it means "use BOTH at once", it may cause synchronization failure if two monitors don’t have the same settings. And 1 on PC means the primary one, 2 is the secondary one. If you have only one monitor on PC, Screen(‘Screens’) = 0;

FrameRate

Returns the value of the nominal frame rate, as measured when you opened the Screen:

```
Framerate = Screen(windowptr, 'FrameRate');
```

MakeTexture

MakeTexture is very much like OpenOffScreenWindow in the old Psychtoolbox 2 in that it produces something that you can copy to the Screen later, and saves it on a “texture” (a buffer). It’s like drawing on a canvas. And once you are done with all the drawing, then you can place the finished product on the Screen.

```
Screen('MakeTexture', WindowIndex, imageMatrix [, optimizeForDrawAngle=0] [,
specialFlags=0] [, floatprecision=0] [, textureOrientation=0] [, textureShader=0]);
```

A texture is like an actual image being stored in memory on the computer and the command uses your video card. There are lots of special things that the computer can do with the texture, in addition to just storing it. For example, you can rotate the texture (by using the drawtexture command), you can have transparent parts of the texture, you can specify how much memory is used for each pixel in the image etc.

MakeTexture has a lot of extra parameters for manipulating things like that, but if you just want it to work like a normal drawing canvas (or off-Screen window), you can just specify a blank white region when you make the texture, like, make a white image with whatever dimensions you want, and then make a texture with that image in it. There might be an easier way, but that would work.

By the way, if you are confused with MakeTexture and DrawTexture, MakeTexture = OpenOffScreenWindow, and DrawTexture = CopyWindow. And for old psychtoolbox users, you may also wonder what is the difference between DrawTexture and Flip, since Flip now is replacing CopyWindow in new psychtoolbox. Here is how: flip makes it so whatever you've drawn (to the off-Screen buffer) gets shown now. Drawtexture is what you use to actually place a texture somewhere on that buffer. For example, if you had an image of a face, you'd load the face into a texture

```
faceTexture = Screen('MakeTexture', w, faceImage)
%faceImage is the name of the matrix containing the face image itself, which you are putting into
%texture faceTexture.
```

then you'd draw the texture to the off-Screen buffer

```
Screen('DrawTexture', w, faceTexture)
```

then you'd show the off-Screen buffer to the participant

```
Screen('Flip',w)
```

MakeTexture is an improved version of OpenOffScreenWindow. Taking a step back, the big difference between the old and new psych toolbox is that now it uses your video card. A video card wants to deal with textures, and wants to flip buffers. It's nice, because it lets us do things like drawdots, which uses the special video card functions.

DrawTexture

This is a complex function, but most of the time you will use it just as in the example above, without the use of the optional inputs. This is the command that allows you to put a Texture that you've made and drawn to into the display buffer (and therefore gets it ready to be shown in the Screen).

```
Screen('DrawTexture', windowPointer, texturePointer [,sourceRect] [,destinationRect] [,rotationAngle]
[, filterMode] [, globalAlpha] [, modulateColor] [, textureShader] [, specialFlags] [, auxParameters]);
```

Draws the texture specified via 'texturePointer' into the target window specified via 'windowPointer'. **'sourceRect'** specifies a rectangular subpart of the texture to be drawn (Defaults to full texture).

'destinationRect' defines the rectangular subpart of the window where the texture should be drawn. This defaults to centered on the Screen.

'rotationAngle' Specifies a rotation angle in degrees for a rotated drawing of the texture
(Defaults to 0 deg. = upright).

'filterMode' How to compute the pixel color values when the texture is drawn magnified, minified or drawn shifted, e.g., if sourceRect and destinationRect do not have the same size or if sourceRect specifies fractional pixel values.
0 = Nearest neighbour filtering,
1 = Bilinear filtering - this is the default.

'globalAlpha' A global alpha transparency value to apply to the whole texture for blending. Range is 0 = fully transparent to 1 = fully opaque, defaults to one.
If both, an alpha-channel and globalAlpha are provided, then the final alpha is the product of both values.

'modulateColor', if provided, overrides the 'globalAlpha' value. If 'modulateColor' is specified, the 'globalAlpha' value will be ignored. 'modulateColor' will be a global color that gets applied to the texture as a whole, i.e., it modulates each color channel.
E.g., modulateColor = [128 255 0] would leave the green- and alpha-channel untouched, but it would multiply the blue channel with 0 - set it to zero blue intensity, and it would multiply each texel in the red channel by 128/255 - reduce its intensity to 50%. The most interesting application of 'modulateColor' is drawing of arbitrary complex shapes of selectable color: Simply generate an all-white luminance texture of arbitrary shape, possibly with alpha channel, then draw it with 'modulateColor' set to the wanted color and global alpha value.

'textureShader' (optional): If you provide a valid handle of a GLSL shader, this shader will be applied to the texture during drawing. If the texture already has a shader assigned (via Screen('MakeTexture') or automatically by PTB for some reason), then the shader provided here as 'textureShader' will silently override the shader assigned earlier. Application of shaders this way is mostly useful for application of simple single-pass image processing operations to a texture, e.g., a simple blur or a deinterlacing operation for a video texture. If you intend to use this texture multiple times or if you need more complex image processing, e.g., multi-pass operations, better use the Screen('TransformTexture') command. It allows for complex operations to be applied and is more flexible.

If you want to draw many textures to the same onScreen- or offScreen window, use the function Screen('DrawTextures'). It accepts the same arguments as this function, but is optimized to draw many textures in one call.

Close and CloseAll

When you are done using a window you created, you have to CLOSE IT to avoid runaway memory problems. If you are re-drawing on your textures or offScreen windows (on different trials for example), I suggest you close all your textures/offScreen windows at the end of each trial, and generate them anew on each trial.
Use 'Close' to close a specific window.

```
Screen('Close', [windoworTextureIndex])
```

For example, after creating image (above) you can close it as follows: Screen('Close',faceTexture).

To close all your windows, simply call: Screen('CloseAll'). This releases almost all the resources taken up by the PsychToolbox in memory.

Flip

Perhaps one of the most fundamental and helpful Matlab functions. Flip front and back display surfaces in sync with vertical retrace and return timestamps.

```
[timestamp StimulusOnsetTime FlipTimeStamp Missed Beampos] =  Screen('Flip', windowPtr [, when] [, dontclear] [, dontsync] [, multiflip]);
```

Parameters inside the function call:
when: you can specify a time, like Screen('Flip',w, GetSecs+1); which would flip the buffer in exactly one second (or as close as possible). You can put [ ], so it ignores time and it just flips as soon as possible. Note that this parameter specifies a time for the system clock, not a relative time (like "one second" meaning "one second from now"). By default, 'Flip' assumes that you mean "the current time on the clock", so if you ignore that parameter, it will flip as soon as it can. But if you want it to happen later, you need to give it "the current time on the clock, plus _something_". GetSecs will give you the current time on the clock.

dontclear: if dontclear = 0 (the default), it clears the background of the next buffer. So when you draw stimuli, it's drawn against a clean/empty background, but if you change it to 1, it leaves the buffer exactly the way it currently is, like it makes a copy. So this makes it that you can have things stay on the Screen

instead of having to draw them every time. For example, let's say you wanted to make a dot on the Screen wherever someone clicks, and you want them to click 10 times instead of having to remember a whole set of locations where they clicked, and add a dot there after each click you can just draw a dot where the mouse is, and flip with the dontclear set to two. This sounds weird, but it's kind of nice. Basically you can do fun animations using flip. You kind of have to play with it to see (see dancingCenter.m from Brian Levinthal). You can also set dontclear to 2. The difference between 1 and 2 is: if you use 1, it ends up doing something like making a copy of the buffer (the way the computer sees it is there are two buffers, one that is on the Screen and one that isn't). So every time you flip, whatever you see on the Screen is also what's on the off-Screen buffer, but if you use 2, you still have two buffers that are independent of each other. They just don't get erased. For example, imagine that you have two images, you copy image 1 to the buffer, and flip (without specifying dontclear), now image 1 is on the Screen, and the other buffer is blank. Next you copy image 2 to the buffer and flip, now you see image 2 on the Screen, and:
if dontclear = 0, the other buffer is now blank;
if dontclear = 1, the other buffer should be image 2;
if dontclear = 2, the other buffer is image 1;

dontclear = 1 means, basically "never change what's on the Screen."
dontclear = 2 means "flip back and forth between the buffers, but never delete them."

dontsync determines whether the program keeps running after matlab sees the flip command, by default it is zero. So if you said Screen('Flip',w,GetSecs+1,[ ],0), matlab would just sit around waiting for a second, and wouldn't do anything else (like checking the keyboard) that you might want it to do. so that could be a problem sometimes. But if you'd like Matlab to 'flip' at a certain time, and you want it to do other stuff while you wait, you change the 0 to a 1. I wouldn't recommend using 2, since it immediately shows the stimulus, even if the monitor isn't ready)

multiflip: defaults to zero. I don't think you'll use "multiflip" unless you have two or more monitors.

PUTIMAGE

This command allows you to copy an image array to a window.

```
Screen('PutImage', windowPtr, imageArray,[rect])
```

Copy "imageArray" to a window, slowly. "imageArray" may be double or uint8. Its values, which should be non-negative integers, are copied to the window, by CopyBits or CopyBitsQuickly. "imageArray" can be an MxN array (for any window) or MxNx3 array (only if the window pixelSize is 16 or 32 bits). In the latter case the three r g b components are combined, as appropriate for the window pixelSize, to form a single value that is copied to the window. In 16-bit mode r, g, and b each contribute 5 bits. In 32-bit mode r, g, and b each contribute 8 bits. The matrix can also be in RGBA mode: RGB matrix with alpha channel (for transparency). This will always be faster using uint8 data type (as opposed to double).

"rect" is in window coordinates. The whole image is copied to "rect", scaling if necessary. The rect default is the imageArray's rect, centered in the window. The orientation of the array in the window is identical to that of MATLAB's numerical array displays in the Command Window. The first pixel is in the upper left, and the rows are horizontal.

Please note that this function is relatively slow and inflexible. Have a look at the 'MakeTexture' and 'DrawTexture' functions for a faster and more flexible way of drawing image matrices.

GETIMAGE

This command allows you to copy into a matrix an image that you drew in one of the PsychToolbox windows (so that you can save it and use it later, or print it…).

```
imageArray=Screen('GetImage', windowPtr, [rect])
```

Slowly copies an image from a window, returning a MATLAB uint8 array. If the window's pixelSize>8 then the returned imageArray has three layers, an RGB image. "rect" is in window coordinates, and its default is the whole window. BEWARE: MATLAB will issue an error if you try to do math on a uint8 array, so you may need to use DOUBLE to convert it, for example:
> imageArray/255
will produce an error, but
> double(imageArray)/255
is ok.

RUSH

```
Rush(string,[priorityLevel])
```

Rush.mex runs a critical bit of your Matlab code with minimal interruption by Macintosh interrupt tasks. The first argument is a string containing Matlab code to be passed to EVAL. Within the string, you can have multiple statements separated by ";" or ",".

Use MaxPriority to determine the highest priority that allows normal operation of the functions you use, e.g. Snd and Screen 'WaitBlanking'. We suggest you always call MaxPriority rather than hard coding any particular priorityLevel, so that your program will gracefully adapt to run optimally on any computer. Here's a typical use:

```
Screen('Screens');        % Make sure all functions (Screen.mex) are in memory.
i=0;                      % Allocate all variables.
loop={
        'for i=1:100;'
                'Screen(window,"WaitBlanking");'
                'Screen("CopyWindow",w(i),window);'
        'end;'
};
priorityLevel=MaxPriority(window,'WaitBlanking');
Rush(loop,priorityLevel);
```

You can also use PRIORITY.

Priority

oldPriority=Priority([newPriority])
Typical use:

```
Screen(window,'WaitBlanking'); % Make sure all RUSHed functions are in memory.
priority(2);
for i=1:100
```

```
        Screen(window,'WaitBlanking');
        Screen('CopyWindow',w(i),window);
end
priority(0);
```

WARNING: Unless you're very careful, using Priority may freeze your keyboard and mouse, leaving you no alternative other than rebooting your computer. Be careful. We strongly suggest that you use the new Rush function instead of Priority, since Rush provides a much safer way of accomplishing the same thing, and more. In any case, your first step should be to read this document, so you'll understand what you're doing.

ScreenTest.m determines, by trial and error, the largest real-time movie you can show. By running at a higher priority you can show a larger real-time movie, without missing any frames.

Time Manager interrupts are blocked by setting the priority to 1 or more. This affects both Time Manager tasks and the Microseconds function. The Microseconds time function continues to advance at the right rate, but in coarse steps of about 0.3 ms, instead of its usual 20 µs, and overflows every 0.1 seconds, if I remember rightly. I think GetSecs uses Microseconds.

For the Windows version of Priority (and Rush), the priority levels set are "process priority levels". There are 3 priority levels available, levels 0, 1, and 2. Level 0 is "normal priority level", level 1 is "high priority level", and level 2 is "real time priority level". Combined with thread priority levels, they determine the absolute priority level of the matlab thread. Threads are executed in a "round robin" fashion on Windows, with the lower priority threads getting cpu time slice only when no higher priority thread is ready to execute. Currently, no tests had been done to see what tasks are pre-empted by setting the Matlab process to real-time priority. It does seem to block keyboard input, though, so for example if you have a clut animation going on at priority level 2, then the force-quit key combo (Ctrl-Alt-Delete) does not work. However, the keyboard inputs are still sent to the message queue, so GetChar or GetClicks still work if they are also called at priority level 2.

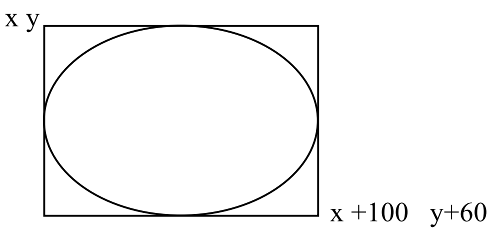
DRAWING with the Psychophysics Toolbox

The following drawing procedures are called within the Screen function and work in very similar ways.

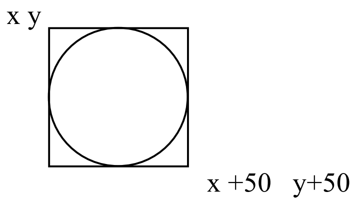
windowPtr points to the Screen you would like to shape to be drawn in.

[color] argument obviously refers to the color you want to draw that particular shape and refers to an clut index or you can specify and [r g b] value. Default is black.

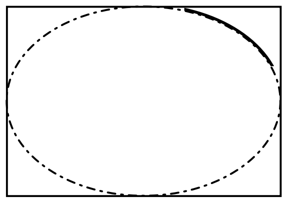
[rect] argument refers to the CIRCUMSCRIBED rectangle around the shape you are drawing. The default [rect] is the entire window. Think of [rect] in the form of [x y +xsize y+ysize], where (x,y) are the coordinates of the LeftTop corner of the rectangle (where your drawing will be tagged to in the target Screen) and xsize and ysize refer to the size along the x and y axis respectively.



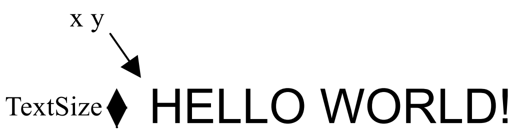
Ovals are circumscribed in rectangles. Here rect = [x y x+100 y+60], where xsize = 100 and ysize =60



Circles are circumscribed in squares, meaning xsize = ysize.



Arcs are drawn within the rectangle that would have circumscribed the entire oval/circle the arc belongs to.



FillRect, FillOval, FillArc

Screen('FillRect',[windowPtr, color],[rect])
Screen('FillOval',windowPtr, [color],[rect])
Screen('FillArc',windowPtr, [color],[rect],startAngle,arcAngle) : angles are measured clockwise from vertical (counterintuitive and counter-mathematical, I know).

DrawLine, FrameArc, FrameRect, FrameOval

Screen('DrawLine',windowPtr, [color],fromX,fromY,toX,toY,[penWidth])
Screen('FrameArc',windowPtr, [color],[rect],startAngle,arcAngle,[penWidth])
Screen('FrameRect',windowPtr, [color],[rect],[penWidth])
Screen('FrameOval',windowPtr, [color],[rect],[penWidth])

[penWidth] is in pixels (default is 1).

FillPoly

Screen('FillPoly',windowPtr, [color],pointList)

pointList is an array with the successive x y coordinates of each corner of the polygon that ends at the initial point. For example, for a ten corner polygon (decagon):

pointList = [X1, Y1; X2, Y2; X3, Y3; …. ; X10, Y10; X1, Y1]

WRITING on your Toolbox Screens

DrawText

Screen('DrawText',windowPtr, textvar, [x], [y], [color] [,backgroundColor] [,yPositionIsBaseline] [,swapTextDirection])

This simple command allows you to write text on your Screen.

textvar is a string variable, e.g., textvar = 'hello world'. Instead of using a variable, you can directly specify a string in quotes:

Screen(image, 'DrawText', 'Hello world', 500, 500, 255)

[x] and [y] are the coordinates of the point in the Screen where you wish to start writing (where you would position your "pen" to start writing). Take note that this is somewhat counter to other Matlab drawing commands, as the text will extend rightwards and <u>upwards</u> from this point (not downwards).

backgroundColor is the color behind the text. By default, text is draw on a transparent background.

"yPositionIsBaseline" If specified, will override the global preference setting for text positioning: It defaults to off. If it is set to 1, then the "y" pen start location defines the base line of drawn text, otherwise it defines the top of the drawn text.

swapTextDirection, if specified and set to 1, sets the writing direction from right to left (as in Hebrew).

If you'd like to specify a font, a size or a style for your text use TextFont, TextSize and TextStyle. Note, you should use this BEFORE you write on the Screen.

TextFont, TextSize, TextStyle

[oldFontName,oldFontNumber]=Screen('TextFont',[windowPtr, fontNameOrNumber]). Default font is system font.

[fontNameorNumber]: numbers change from computer to computer, better use font names (e.g. 'Helvetica', 'Times New Roman', …).

Screen('TextSize', windowPtr, [fontSize])

[fontSize]: Number of pixels the tallest letter will extend vertically.

Screen('TextStyle',windowPtr, [style])

[style]: 0=normal,1=bold,2=italic,4=underline.

Usage:

```
Screen('TextFont',window, 'Times New Roman');
Screen('TextSize',window, 48);
Screen('TextStyle',window, 1);   % for bold text
Screen('DrawText',window, 'Hello world.', 100, 100,255);
```

Also useful is to know how large your text will be on the Screen, use TextBounds:

[normBoundsRect, offsetBoundsRect] = Screen(**'TextBounds'**, windowPtr, textvar); %

Accepts a window pointer and a 'text' string. Return in 'normBoundsRect' a rect defining the size of the text in units of pixels. Returns in 'offsetBoundsRect' offsets of the text bounds from the origin, assuming that the text will be drawn at the current position of the text drawing cursor. Only the default high quality text renderers returns a perfect bounding box. The optionally selectable low-quality, fast renderers on Windows and Linux return a bounding box which doesn't take letters with descenders into account - Descenders are outside the returned box.

You can also use DrawText to get the location of your pen after drawing (newX, newY). Call it using two outputs:

[newX,newY] = Screen(windowPtr,'DrawText',text,[x],[y],[color])

GETTING RESPONSES

KbCheck

This function returns the status of the keyboard (whether a key has been pressed), when that happened and which key it was. Usage:

[keyIsDown, secs, keyCode] = KbCheck

keyIsDown	1 if any key, including modifiers such as <shift>, <control> or <caps lock> is down. 0 otherwise.
secs	time of keypress as returned by GetSecs.
keyCode	A 256-element array. The first 128 elements correspond to the ascii sequence of the characters.

KbWait

This functions waits until any key is down and returns the time (GetSecs).
Command-Period always causes an immediate exit. Usage:

Secs = KbWait

WARNING: Hitting CapsLock makes KbCheck and KbWait think that you're holding the shift key down. They will continue to think so (returning 1) until you release the shift by hitting CapsLock again.

NOTE ON SPEED: KbCheck and KbWait are MEX files, which take time to load when they're first called. They'll then stay loaded until you flush them (e.g. by changing directory or calling CLEAR MEX).

NOTE ON UNCERTAINTY: This call has an uncertainty determined by the frequency at which your operating system polls the keyboard. It is on the order of 11 ms for Macs. I use fast keyboards (checked by Windows every millisecond).

GetChar
Waits (if necessary) for a typed character and return it. Usage:

```
[char,when] = GetChar
```

CharAvail
Returns 1 if a character is available in the event queue, 0 if not. Note that this routine SHOULD leave the character in the queue, but sometimes steals it (humm…). Call GetChar to remove the character from the event queue. Usage:

```
avail = CharAvail
```

NOTE on SPEED: KbCheck and KbWait are event oriented and very fast, whereas GetChar and CharAvail are character oriented and slower. CharAvail and GetChar call the Event Manager, which allows the system to get control. Sometimes CharAvail will take tens of milliseconds to return, so don't use CharAvail in real-time loops. And there can be some delay between when the key is pressed and when CharAvail or GetChar detects it. If precise timing of the keypress is important, use KbCheck or KbWait.

If only a meta key was hit (SHIFT, ALT, CTRL), KbCheck will return true, because a key was pressed, but CharAvail will return false, because no character was generated.

FlushEvents can be used to clear the character event buffer read by GetChar. Unlike GetChar, KbCheck only reports keys depressed at the moment KbCheck is called. FlushEvents has no effect on KbCheck.

FlushEvents
Removes all events of the specified types from the event queue. The arguments can be in any order. Empty strings are ignored. Usage:

```
FlushEvents(['mouseUp'],['mouseDown'],['keyDown'],['autoKey'],['update'],...)
```

EventAvail

Returns 1 if an event of the requested type(s) is available in the event queue, and 0 otherwise. You must supply at least one eventType. If you supply more than one eventType, then you may want to use the optional second output argument to know which eventType EventAvail found (the highest in the event queue). Usage:

```
[isAvail,eventType]=EventAvail(['mouseUp'],['mouseDown'],['keyDown'],['autoKey'],['update
'],...)
```

CharAvail returns EventAvail('keyDown').

To test for mouse clicks, use EventAvail('mouseDown','mouseUp')

GetClicks

Wait for the user to click the mouse, count the number of clicks that occur within a system inter-click interval of each other, and then return the number of clicks and the mouse location. Usage:

```
[clicks,x,y] = GetClicks([windowPtrOrScreenNumber])
```

The x,y location is the location at the downstroke of the first mouse click. The mouse position (x,y) is "local", i.e. relative to the origin of the window or Screen, if supplied; otherwise it's "global", i.e. relative to the origin of the main Screen.

The allowed inter-click interval can be adjusted in the Mouse Control Panel: "double-click speed".

If a sound was playing (see Snd), it is stopped at the first mouse click.

GetMouse

Returns the current (x,y) position of the cursor and the up/down state of the mouse buttons. Usage:

```
[x,y,buttons] = GetMouse([windowPtrOrScreenNumber])
```

"buttons" is a 1xN matrix where N is the number of mouse buttons. Each element of the matrix represents one mouse button. The element is true (1) if the corresponding mouse button is pressed and false (0) otherwise. There may be up to 3 buttons on your mouse (maybe more!).

To test your mouse:

```
[x,y,buttons] = GetMouse;

% Test if any mouse button is pressed.

 if any(buttons)
   fprintf('Someone''s pressing a button.\n');
  end

  % Test if the first mouse button is pressed.

 if buttons(1)
   fprintf('Someone''s pressing the first button!\n');
  end
```

```
% Test if the second mouse button is pressed.
if length(buttons)>=2 & buttons(2)
  fprintf('Someone''s pressing the second button!\n');
end
```

length(buttons) tells you how many buttons there are on your mouse.

NOTE: If you use GetMouse to wait for clicks, don't forget to wait for the user to release the mouse button, ending the current click, before you begin waiting for the next mouse press.

```
fprintf('Please click the mouse now.\n');
[x,y,buttons] = GetMouse;
while any(buttons) % if already down, wait for release
        [x,y,buttons] = GetMouse;
end
while ~any(buttons) % wait for press
        [x,y,buttons] = GetMouse;
end
while any(buttons) % wait for release
        [x,y,buttons] = GetMouse;
end
fprintf('You clicked! Thanks.\n');
```

SetMouse

Move the mouse cursor to the passed (x,y) position. (x,y) are absolute positions on the main Screen. Usage:

```
SetMouse(x,y,)
```

Cursor updating is usually carried out by a System task that runs once per tick, so it's likely that SetMouse doesn't take effect until the next tick. Instead of the elaborate check suggested above, it might be enough, after calling SetMouse, to simply WaitSecs(0.002) before calling GetMouse to be sure of getting the new position (a 2 milliseconds wait should be long enough to cover a system tick, which theoretically should last about 1 ms).

Use **HideCursor** and **ShowCursor** to hide/show the mouse from your main window.

WaitSecs

Waits "s" seconds (with high precision). WaitSecs(s) is similar to Matlab's built-in PAUSE(s) command. The advantage of WaitSecs(s) is that it is much more accurate, although PAUSE can be turned 'ON' and 'OFF', which is useful for scripts.

> NOTE ON SPEED: the first time you access any MEX function or M file, Matlab takes several hundred milliseconds to load it from disk. Allocating a variable takes time too. Usually you'll want to omit those delays from your timing measurements by making sure all the functions you use are loaded and that all the variables you use are allocated, before you start timing. MEX files stay loaded

until you flush the MEX files (e.g. by changing directory or calling CLEAR MEX). M files and variables stay in memory until you clear them.

GetSecs

This is the function you'll use for your timing. It returns the time in seconds (with high precision) since the computer started. Usage:

```
secs = Getsecs
```

So to time an event:

```
t1 = Getsecs;
{code relating to event being timed}
t2 = GetSecs;
EventDuration = (t2 – t1) * 1000;   % for a value in milliseconds
```

GetSecsTick

GetSecsTick returns the fraction of a second which is a tick of the GetSecs clock. It is the precision of your GetSecs function. Usually, in the order of microseconds.