I/O in Matlab

I/O can be performed to/from the command window or to/from a disk or some other secondary storage device.

Non-File Unformatted I/O in MatLab

- `disp(X)` displays the value of variable `X` without printing the variable name.
- Another way to display a variable is to type its name (without a semi-colon), i.e. `X`, but this displays a leading string, i.e. **X =**, above the values of **X**, which may not be wanted. Non-use of semicolon to display a variable is often called **echoing** that variable.

- Beware of large variables. For example, `I=imread('lena.jpg')` without a semicolon causes $512 \times 512 = 262,144$ integers (unsigned characters) to be printed on the command window!

- The `input` command can be used to prompt the user for input. For example, `x=input('Input the value of x: ');` will print the string **Input the value of x:** on the command line and wait for the user to type a value followed by pressing the **return** key.
- Another useful example:

```
prompt = 'Do you want more? Y/N [Y]: ';
str = input(prompt,'s');
if isempty(str)
    str = 'Y';
end
```

If the input in `str` is empty, this code assigns a default value, `'Y'`, to the output string, `str`.

File Processing in MatLab

- To use files in MatLab we need some way to select a desired file and to read from and write to it. MatLab uses a **file id**, a positive integer that is assigned to a file when it is opened. When we close a file, this number/file association is deleted.

- Data can be written/read from files in 2 ways: as binary data (in general not readable by humans) or as formatted character data (this ascii data is readable by humans). Binary I/O is much faster that ascii I/O.

- If you know C then I/O commands in Matlab are somewhat easier to understand (excepted for their vectorized use, as will be explained below).

- The table below gives a summary of the I/O functions in MatLab:

Category	Function	Description
Load/save workspace	load	load workspace
	save	save workspace
File Opening/closing	fopen	open file
	fclose	close file
Binary I/O	fread	reads binary data from file
	fwrite	write binary data to file
Formatted I/O	fscanf	reads formatted data from file
	fprintf	write formatted data to file
	fgetl	read line from file, discard newline character
	fgets	read line from file, keeps newline character
File positioning, status and miscellaneous	delete	deletes file
	exist	does file exist?
	ferror	inquire file error status
	feof	test end of file
	fseek	set file position
	ftell	check file position
	frewind	go back to file beginning
Temporary files	tempdir	get temporary directory name
	tempname	get temporary file name (uses a universal unique identifier as the file number)

Save and Load Commands in MatLab

- Use **save(filename)** [function line] or **save filename** [command line] versions of **save** to save all variables from the current workspace as a MatLab formatted binary file (MAT-file) called filename. If filename already exists, **save** overwrites the file.
- **save filename variables** saves only the listed **variables** in the MAT file.
- **save filename variables -append** adds new variables to an existing file, and does not overwrite it.
- **save filename variables fmt** saves in the file format specified by fmt. The default value for **fmt** is -mat, but other values can be **-double** and **-ascii** or **-acsii, -double**.

- **save(filename)** is the function form of the syntax for save. For example, the following pair of commands are equivalent:

```
save('test.mat','x','y','z')   % function form to save all variables
save test.mat x y z            % command form to save all variables
```

- The **load(filename)** or **load filename** command loads MAT data from filename.
  - If the filename is a MAT-file, then **load(filename)** loads variables in the MAT-File into the MatLab workspace.
  - If filename is an Ascii-file, then **load(filename)** creates variables containing data from the file.
  - Look in the workspace to see the name and values of variables loaded by a load command.

- For example, to load a file named data.mat, these statements are equivalent:

- For example, to load the variable named **x** from data.mat we can use:

```
load('data.mat','x','y','z')  % function form
load data.mat x y z           % command form
```

- **load(filename,'-ascii')** treats **filename** as an ASCII file, regardless of the file extension.

- **load(filename,'-mat')** treats filename as a MAT-file, regardless of the file extension.

- **load(filename,'-mat',variables)** loads the specified variables from the specified MAT-file.

- We can use **whos** in a MAT-file, data.mat. For example:

```
matObj = matfile('data.mat');
whos(matObj)
```

creates a MatLab object from data.mat and **whos** prints the Name, Size, Number of Bytes, Class and Attributes of each variable in data.mat:

```
Name                Size                Bytes  Class     Attributes

x_array           180x360               518400  double
x_int               1                   8       double
```

Note that `x_array` is a 180×360 array of doubles that has 518400 bytes (180*360*8) bytes while `x_int` is an integer variable (8 bytes).

- Lastly, `save` by itself saves all variables in the workspace to file **matlab.mat**. Similarly, `load` by itself loads all variables in the file **matlab.mat** into the workspace.

Opening/Closing a File in MatLab

- To read data from a file, we must first open that file. To open a file for reading, we can use

```
fid=fopen(filename,permission,machinefmt,encodingIn)
```

- **fid=fopen(filename)** opens the file, named filename, for binary read access, and returns an integer value to **fid** equal to or greater than 3. MatLab reserves file identifiers 0, 1, and 2 for standard input, standard output (the screen), and standard error, respectively. If **fopen** cannot open the file it returns -1 as the value of **fid**.

- **fid=fopen(filename,permission)** opens the file with the type of access specified by permission. Some permission values can be strings com-

posed of 'r' (read), 'w' (write), 'a' (append), 'r+' (open existing file for read/write), 'w+' (create a new file for read/write).

- **fid=fopen(filename,permission,machinefmt,encodingIn)** opens the file with **permissions** as above, with **machinefmt** being one of 'n' (native) [the default byte ordering for your computer], 'b' (big endian), 'l' (little endian), 's' (big endian, 64 bits) and 'a' (little endian, 64 bits). By default, all existing platforms supported by MatLab use little endian ordering for new files. The **encodingIn** option allows the user to specify the type of character encoding to be used for subsequent read and write operations, including **fscanf**, **fprintf**, **fread** and **fwrite**, specified as one of the following strings (among others) of 'UTF-8' or 'ISO-8859-1' or 'windows-1252' or 'macintosh' or 'US-ASCII'.

- For example `fid=fopen('data.txt','w','n','macintosh')` opens file `data.txt` for writing in native format using macintosh character encoding.

- Some examples:

  1. `fid=fopen('example.dat','r')` opens file `example.dat` for reading only ('r').
  2. `fid=fopen('example.dat','w')` opens file `example.dat` for writing only ('w').
  3. `fid=fopen('example.dat','a')` opens file `example.dat` for appending ('a').
  4. `fid=fopen('myfile','r+')` and

`fid=fopen('myfile','w+')` both open file **myfile** for binary input and output: the first statement requires the file to exist before it is open while the second statement deletes any existing file.

- The **fclose** function closes a file.
- `status=fclose(fid)` closes the file with file identifier `fid`. `status` is 0 for a successful closing and -1 for an unsuccessful closing.
- `status=fclode('all')` closes all open files except for `stdout` (fid=1) and `stderr` (fid=2). Again, `status` is either 0 or -1.

Binary I/O

- The `fwrite` function has format:

  ```
  count=fwrite(fid,array,precision),
  ```

  where `fid` is the file id of a file that has been opened with `fopen`, `array` is the array of values to be written (don't forget that a single variable is actually a 1×1 array) and `count` is the number of values written.

- Remember that MatLab uses column major storage, do the data is written column by column.

- `precision` are strings such as 'char', 'uchar', 'int32', 'uint64', 'float32', and 'float64'.

- The `fread` function has the format:

  ```
  [array,count]=fread(fid,size,prevision),
  ```

  where `fid` is the file id of an opened file and `size` is the number of values to be read.
- `percision` string has the form

  ```
  disk_precision => array_precision,
  ```

  where `disk_precision` specifies the precision of the data on the disk and `array_precision` specifies the precision of the data of the array. For example `single=>single` reads data in single precision from the disk and stores it in a single precision array while `single=>double` reads data in single precision from the disk and stores it in a double precision array. Type conversions that make sense are allowed.

- There are 3 possible values for the `size` argument:

  1. `n` means read exactly `n` values (`array` is a column array containing n values.
  2. `Inf` means read until the end of the file. Now `array` is a column array containing all the values in the file.
  3. `[n m]` reads exactly $n \times m$ values into a $n \times m$ array. Note `[n inf]` means read n columns of data until the end of the file.

Formatted I/O

- We present formatted MatLab I/O functions in this section.
- The **fprintf** statement writes formatted data to a file as:

  ```
  count=fprintf(fid,format,val1,val2,...)
  ```

  where `fid` if the file id of the file the data is written and `format` is the character string controlling the appearance of the data.
- If `fid` is missing the data is written to the command window.
- In `format` there are (usually) one of more **format specifiers**. A format specifier has the following components:

  1. Marker (required) - %, use %% to print a single %.

2. Modifier (optional) - a flag `'-'` left justifies an argument, a flag `'+'` requires a plus or minus symbol to be printed for a number and a flag `'0'` causes the argument to padded with leading zeros on the left. decimal points and signs are counted as characters.
3. Field width (optional) - specifies the total number of characters to be used for the formatted value.
4. Precision (optional) - the number of digits to the right of the decimal point.
5. Format descriptor - for example, %s for string, %e for exponential number, %d for integer number, %f for a floating point number (single or double) and %g for matching any type.
6. Some escape character that can be used in a format string include

`\n` (newline), `\t` (tab) and `\\` (print a backslash character).

- Some examples:

```
fprintf('%6d %d %06d %-6d\n',123,123,123,123);
```

prints

```
___123 123 000123 123____
```

while

```
fprintf('%f %8.2f %10.3e\n',123.456,123.456,123,456);
```

prints

```
123.456 __123.46 _1.235e+02
```

while

```
fprintf(%s %8s %-8s\n','john','john','john');
```

prints

```
john ____john john____
```

The underscore characters (_) indicate blank characters.

- **sprintf** is the same as **fprintf** except it writes its formatted data to a character string instead of a file (or the command window).
- For example, `str=sprint('%6.2f',123.456)` sets `str` to '123.46'.
- `sprintf` can be used in place of `num2str` to convert numbers into character strings.
- **fscanf** reads formatted data from a file:

```
[array,count]=fscanf(fid,format,size)
```

where `fid` is the file id of the file the data is to be read from, `format`

is the format string controlling how the data is to be read. `array` is the array that receives the data and `count` is the number of data items read.

- `size` can be one of:

  1. `n` reads exactly `n` values (`array` is a column vector)
  2. `inf` reads until the end of file (`array` is a column vector with all this data)
  3. `[n m]`a reads $n \times m$ values into a $n \times m$ array.

- Suppose a file has 2 lines of data:

```
10.00     20.00
30.00     40.00
```

- `[z,count]=fscanf(fid,'%f')` sets `size` to 4 and `z` to the column vector $\begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \end{bmatrix}$.
- `[z,count]=fscanf(fid,%f',[2 2])` sets `size` to 4 and `z` to the 2×2 array `z` to the 2×2 array $\begin{bmatrix} 10 & 30 \\ 20 & 40 \end{bmatrix}$
- The **line=fgetl(fid)** reads the next line of a file (excluding the end-of-line character) into variable **line**. When the end of the file is encountered line is set to -1.
- The **line=fgeta(fid)** reads the next line of a file (including the end-of-line character) into variable **line**. When the end of the file is encountered line is set to -1.

- Another I/O example:

```
fid = fopen('data.txt');} followed
A = fscanf(fid, '%g %g', [2 inf]);
fclose(fid);
```

opens file `data.txt` for reading, read 2 values at a time from the file pointed to be **fid** until the end of file is encountered. Format **%g %g** allows any type of data to be read. The results of this read are stored in array `A`. **[2 inf]** means read 2 columns of data from **data.txt** until the end of the file.

Formatted versus Binary I/O

- Formatted files can display data on output devices, can transport data between different computers with different architectures, require relatively large disk space, are slow (requires a lot of computer time) and have possible truncation/roundoff error caused by formatting.

- Binary files cannot display data on output devices, cannot easily transport data between computers with different architectures, requires relatively small disk space, are fast (little computer time required), and has no truncation/roundoff error.

- In general, binary I/O is upto 100 times faster than formatted I/O!!!

File Positioning and Status MatLab Functions

- In general, MatLab files are accessed sequentially, i.e. they are read from the front to the end. However,sometimes, we need to do **random** access into a file. The function **exist** can determine whether or not such a file exists, **feof** and **ftell**, tell you where you are in a file and **frewind** and **fseek** let you move around in the file. **ferror** provides a detailed description of the cause of errors when they occur.

- The function **exist** checks for the existence of a variable in the workspace, a built-in function or a file in the MatLab search path:

```
ident=exist('item','kind')
```

- If `'item'` exists then a value is returned depending on its type:

0	item not found
1	item is a variable in the workspace
2	item is a m-file (or of unknown type)
3	item is a mex-file
4	item us a mdl file
5	item is a built-in function
6	item is a p-file
7	item is a directory
8	item is a java class

- The `'item'` value is restricted if `kind` has a value of `'var'`,`'file'`, `'builtin'` or `'dir'`.

- The **exist** function allows the users to check if a file exists or not before it is overwritten by an **fopen** command.

- The **ferror** command translates the error indicator into an easy-to-understand character message:

  ```
  [message,errnum]=ferror(fid)
  ```

  returns the most recent file processing error message. If a file operation was successful the message is `'..'` (and the error number is 0).

- The **feof(fid)** function tests if the current file pointer is at the end of the file.

- `position=ftell(fid}` returns a non-negative integer telling you where the file pointer is. A value of -1 means `ftell` was unsuccessful (use **ferror** pointer to find out why).

- `frewind(fid)` allows the user to reset the file pointer to the beginning of the file (offset 0). The name of the function comes from the old days when tapes instead of disks were used.

- The **fseek** command allows the user to set a file position indicator to an arbitrary position within the file. It has form:

  ```
  status=fseek(fid,offset,origin),
  ```

  which sets the file pointer in file `fid` to be `offset` bytes from the `origin`. A positive `offset` number means move `offset` bytes from

the current position towards the end of the file. A negative `offset` numbers means move `offset` bytes towards the beginning of the file.

- The `origin` variable can have 1 of 3 values:

  1. `'bof'` (beginning of the file),
  2. `'cof'` (current position in the file) and
  3. `'eof'` (end of the file).

- If `status` is 0 the file operation was successful. If `status` is -1 use **ferror** to determined why the request failed.

- Consider the following MatLab segment:

```
[fid,msg]=fopen('file.dat','r');
status=fseek(fid,-10,'bof');
if(status~=0)
   msg=ferror(fid);
   disp(msg);
end
```

This segment open file `'file.dat'` and attempts to set the file pointer to 10 bytes before the beginning of the file. This is impossible and **fseek** returns a value of -1 and **ferror** gets the appropriate error message:

```
Offset is bad - before beginning of file.
```