MATLAB Week 2

10 November 2009

Outline

- Discuss Arrays, some syntax and built in functions all together
  - Tidbits of syntax and built in functions come up while learning arrays
- More syntax on flow control of program (i.e. for, while, if, switch)
- File I/O
  - Text, binary, netCDF, HDF

Arrays

- MATLAB is adept at handling arrays
  - Optimized for vector/matrix operations
  - This allows for a reduction of code in some cases
- Array types: Numeric, character, logical, cell, structure, function handle
  - Numeric types: single, double, int8, int16, int32, uint8, uint16, uint32

Numeric Arrays

- One dimensional arrays, called vectors
  - Can create row or column vectors
  - Row vector
    - A few ways to create a row vector

```
Command Window
>> foo = [3 4 5 6]

foo =

     3     4     5     6
```

```
>> foo = [3,4,5,6]

foo =

     3     4     5     6
```

- Or use ":"

Numeric Arrays

- Column vectors
  - A few ways to generate column vectors too
  - Semicolon between elements starts new row
  - Transpose a row vector, or use return between elements

```
>> bar = [3;4;5;6]

bar =

     3
     4
     5
     6

fx >>
```

```
>> bar = [3,4,5,6]'

bar =

     3
     4
     5
     6
```

```
>> bar = [3
4
5
6]

bar =

     3
     4
     5
     6
```

Numeric Arrays

- You can also append vectors into one big row or column vector
- Use built in function to generate vectors
  - **linspace**
    - Create a row vector of linearly spaced elements
  - **logspace**
    - Create a row vector of logarithmically spaced elements

Numeric Arrays

- Two-dimensional arrays, called a matrix in MATLAB often
- Size = rows by columns
  - [r c] = size(array_name) if array_name is a matrix
  - **size** will work for n-dimension arrays and output size of each dimension into a row vector
- Basic matrix creation:

```
>> foo = [3,4,5;6,7,8]

foo =

     3     4     5
     6     7     8
```

Numeric Arrays

- Array addressing
  - Vector: foo
    - foo(:) gives all row or column elements
    - foo(1:5) gives the first five row or column elements
    - foo(2) gives the second element
  - Matrix: bar
    - bar(1,3) gives the first row, third column element
    - Bar(:,2) gives all elements in the second column
    - Bar(1,:) gives all elements in the first row
    - Bar(3:4,1:3) gives all elements in the third and fourth rows that are in the first through third columns

Numeric Arrays

- Array operations
  - MATLAB has array or element by element and matrix operations when dealing with arrays
- Element-by-element operations
  - Multiplying a vector or array by a scalar is an easy example

```
foo =

     3     4     5
     6     7     8

>> foo*2

ans =

     6     8    10
    12    14    16
```

Numeric Arrays

- How do you multiply to arrays element-by-element?
  - Do it in a for loop
    - Very slow, avoid whenever possible
  - Use element-by-element operations
  - Addition and subtraction are the same: +, -
  - Multiplication, right and left division and exponentiation use "." before the symbol
    - i.e. foo.*bar or foo./bar or foo.^3

Numeric Arrays

- Examples of element-by-element operation results

```
>> foo = [1 2; 1 2];
>> bar = [3 4; 3 4];
>> foo.*bar

ans =

     3     8
     3     8

>> foo./bar

ans =

    0.3333    0.5000
    0.3333    0.5000
```

```
>> foo+bar

ans =

     4     6
     4     6

>> foo.^2

ans =

     1     4
     1     4
```

Numeric Arrays

- Matrix operations
  - Matrix operations are the same symbols as standard scalar operations
  - Apply when multiplying or dividing matrices

```
>> col = [1 2 3 4]'

col =

     1
     2
     3
     4

>> row = [1 2 3 4]

row =

     1     2     3     4

>> row*col

ans =

    30
```

```
>> col*row

ans =

     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16
```

```
>> foo*bar

ans =

     9    12
     9    12
```

Numeric Arrays

- Useful built-in functions for working with arrays
  - Size, linspace and logspace were already mentioned
  - **Num_of_elements = length(array)**
    - Returns the length of the longest dimension
  - **Max_val = max(array)**
    - Returns the largest element in the array if it is a vector
    - Returns a row vector of the largest element if the array is a matrix
  - **min_val = min(array)**
    - Same as **max**, but returns the minimum values

Numeric Arrays

- **sum(array)**
  - Returns the sum of a vector or a row vector of the sum of each column
- **find(array)**
  - Returns an array with the locations of the nonzero elements of the array
  - Will talk about find more in week 4

Numeric Arrays

- When dealing with numeric arrays, the **clear** function can come in handy
  - If you have **foo** declared as a 2x2 array
  - Then create two 5x1 arrays, **x** and **y**
  - Try to do: **foo(:,1) = x**
  - Produces an error because MATLAB thinks **foo** should be a 2x2 array and **x** won't fit into the first column of **foo**
  - Use clear to reset **foo**

Numeric Arrays

- MATLAB can create several special matrices
  - Identity, zeros and ones
  - Useful for initializing matrices to ones, zeros or when you may need to use the identity matrix
- MATLAB also has functions to check for NaN, inf, or if an array is empty
  - **isnan**, **isempty**, **isinf**

Cell Arrays

- A cell array is an array where each element can contain another array with different dimensions if needed
- Cell arrays can hold different classes of arrays
- Cell arrays aren't used too much in most MATLAB sessions
  - I've only used them for one type of FILE I/O
  - They are used in some toolboxes too

Cell Arrays

- Creating a cell array is very similar to creating a numerical array
- To create a cell array, use {} instead of []

```
>> A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
>> A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi}

A =

        [3x3 double]    'Anne Smith'
    [3.0000 + 7.0000i]    [1x9 double]
```

- Cell arrays can also be created one element at a time or pre-allocated and then defined

Cell Arrays

- Indexing a cell array is more complex than a numeric array
- To display a list of each cell and the type of array in each cell: **A(:)**
  - **cellplot(A)** will give a graphical representation of the cell array
- To display the contents of each cell: **A{:}**
- To go into an array in a given cell location use a combination of () and {}
  - **A{1}(2,2)** will return the element at (2,2) for the array in the first cell of **A**

Structures

- Structures are a MATLAB data type used to store different types of data in a single unit
- A structure consists of fields
- Each field contains an array of some MATLAB type
- Actually similar to a cell array in that it combines different sized arrays of various data types into one entity

Structures

- Two ways to create a structure in MATLAB
  - Define each field of a structure individually
    - Structure_name.field_name

```
>> s.a = 10;
>> s.b = 4;
>> s.c = 'abc';
>> s

s =

    a: 10
    b: 4
    c: 'abc'
```

  - Use the **struct** function call

```
>> q = struct('a',10,'b',4,'c','abc')

q =

    a: 10
    b: 4
    c: 'abc'
```

Structures

- One more example

```
>> s.a = [1 2 3 4 5];
>> s.b = 'the dog walks';
>> s.c = eye(3);
>> s

s =

    a: [1 2 3 4 5]
    b: 'the dog walks'
    c: [3x3 double]
```

Structure Arrays

- MATLAB can create arrays of structures
  - Just add in the index of the array location before the field name

```
>> st(1).a = 'a';
>> st(2).a = 'A';
>> st(1).b = 'b';
>> st(2).b = 'B';
>> st

st =

1x2 struct array with fields:
    a
    b
```

  - Can create n-dimension structure arrays

Structure Arrays

- Indexing structures and structure arrays can get complicated
  - You need to specify location in structure array, field name and field location
  - Structures can contain cell arrays, further complicating things
  - Nested structures are possible too

Structure Arrays

- Some simple examples

```
>> st = struct('name',{'bob' 'tom' 'bill'},'age',{2 3 4},'other',{[1 2 3 4] [4 5 6] [6 3 1]})

st =

1x3 struct array with fields:
    name
    age
    other
```

```
>> st(1).name

ans =

bob

>> st(2).name

ans =

tom
```

```
>> st(2).name(1)

ans =

t

>> st(2).name(1:3)

ans =

tom
```

```
>> st(2).other(2)

ans =

     5

>> st(2).other

ans =

     4     5     6
```

- See MATLAB help for much more in-depth discussion on structures or cell arrays for that matter

Strings

- Strings are just character arrays
  - Can be multidimensional
  - However, then each string is required to be the same length to keep array rectangular
  - Cell arrays are handy for multiple strings of different length
    - File I/O of characters (discussed later today)

Strings

- Examples
  - We've already seen many examples scattered throughout the first two lectures, but here a few more

```
phrase =

a bird in hand is worth two in the bush

>> phrase(1:6)

ans =

a bird
```

```
>>
>> string_array = zeros(5,5);
>> string_array(:,1) = 'abcde';
>> string_array(:,2) = 'buddy';
>> string_array(:,3) = 'do';
??? Subscripted assignment dimension mismatch.

>> string_array(:,3) = 'dooooo';
??? Subscripted assignment dimension mismatch.
```

Strings

- Useful functions
  - String manipulation: **sprintf**, **strcat**, **strvcat**, **sort**, **upper**, **lower**, **strjust**, **strrep**, **strtok**, **deblank**, etc.
  - String comparisons
    - Should use **strcmp** when comparing strings
    - If you have two strings **A** and **B**, **A==B** will give a row vector of results for each character
    - **Strcmp** will compare the entire string, or a subset if you want

M-Files

- M-files are essentially script files where you can place a bunch of MATLAB commands and execute the m-file repeatedly
  - Can have everything in an m-file, function calls, variable allocation, function definitions, figure generation and modification, etc
- Nearly everything will probably be done in an m-file, rather than at the command prompt

M-Files

- MATLAB has the editor window for creating, editing, debugging, running and saving m-files
- MATLAB color codes m-files for easier reading
- There is also real-time syntax checking
  - Essentially spell check for code

M-files

- Example m-file

Editor - C:\Users\Andy\Documents\MATLAB\first_m_file.m

File Edit Text Go Cell Tools Debug Desktop Window Help

Stack: Base

```
%the first m-file
%comments are green

%a good first statement is
clear all;
%this clears the workspace and any figures as well


a = 4:7;
b = [1 2 3 4];

c = a.*b;

string = 'name';
%strings are colored light purple

%reserved words are colored blue
if(a(1) == 4)
    d = 6;
end

%matlab has real-time syntax checking
a = (b-d;
```

edit_input_vortex_v2.m | first_m_file.m

Program Flow Control

if

Execute statements if condition is true

Syntax

```
if expression, statements, end
```

Description

`if expression, statements, end` evaluates *expression* and, if the evaluation yields logical `1` (`true`) or a nonzero result, executes one or more MATLAB commands denoted here as *statements*.

expression is a MATLAB expression, usually consisting of variables or smaller expressions joined by relational operators (e.g., `count < limit`), or logical functions (e.g., `isreal(A)`). Simple expressions can be combined by logical operators (`&&`, `||`, `~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

```
(count < limit) && ((height - offset) >= 0)
```

Nested `if` statements must each be paired with a matching `end`.

The `if` function can be used alone or with the `else` and `elseif` functions. When using `elseif` and/or `else` within an `if` statement, the general form of the statement is

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

See Program Control Statements in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be true or nonzero for the entire expression to be considered true. For example, the statement `if (A < B)` is true only if each element of matrix `A` is less than its corresponding element in matrix `B`. See Example 2, below.

Program Flow Control

switch

Switch among several cases, based on expression

Syntax

```
switch switch_expr
 case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

Discussion

The `switch` statement syntax is a means of conditionally executing code. In particular, `switch` executes one set of statements selected from an arbitrary number of alternatives. Each alternative is called a *case*, and consists of

- The `case` statement
- One or more case expressions
- One or more statements

In its basic syntax, `switch` executes the statements associated with the first case where *switch_expr* == *case_expr*. When the case expression is a cell array (as in the second case above), the *case_expr* matches if any of the elements of the cell array matches the switch expression. If no case expression matches the switch expression, then control passes to the `otherwise` case (if it exists). After the case is executed, program execution resumes with the statement after the `end`.

The *switch_expr* can be a scalar or a string. A scalar *switch_expr* matches a *case_expr* if *switch_expr*==*case_expr*. A string *switch_expr* matches a *case_expr* if `strcmp(switch_expr, case_expr)` returns logical `1` (`true`).

> **Note for C Programmers** Unlike the C language `switch` construct, the MATLAB `switch` does not "fall through." That is, `switch` executes only the first matching case; subsequent matching `case`s do not execute. Therefore, `break` statements are not used.

Program Flow Control

- **while**
- Repeatedly execute statements while condition is true
- **Syntax**
- while *expression*, *statements*, end

- **Description**
- while *expression*, *statements*, end repeatedly executes one or more MATLAB *statements* in a loop, continuing until *expression* no longer holds true or until MATLAB encounters a break, or return instruction. thus forcing an immediately exit of the loop. If MATLAB encounters a continue statement in the loop code, it immediately exits the current pass at the location of the continue statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *statements*.
- *expression* is a MATLAB expression that evaluates to a result of logical 1 (true) or logical 0 (false). *expression* can be scalar or an array. It must contain all real elements, and the statement all(A(:)) must be equal to logical 1 for the expression to be true.
- *expression* usually consists of variables or smaller expressions joined by relational operators (e.g., count < limit) or logical functions (e.g., isreal(A)). Simple expressions can be combined by logical operators (&&, ||, ~) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to Operator Precedence rules.
- (count < limit) && ((height - offset) >= 0)*statements* is one or more MATLAB statements to be executed only while the *expression* is true or nonzero.
- The scope of a while statement is always terminated with a matching end.

Program Flow Control

- **For Loop**
- Iteratively executes child components
- **Description**
- This component functions like the MATLAB for loop, except that instead of executing a statement, it executes its child components. It must have at least one child component to execute.
- **Loop Type**
- The loop type can have incremented indices or a vector of indices. For more information on for loops and indices, see for in the MATLAB documentation.
- **Incremented indices**: Executes a for loop of the form:
- for varname=x:y:z
  - **Start**: Corresponds to x in the previous expression.
  - **Increment**: Corresponds to y in the previous expression.
  - **End**: Corresponds to z in the previous expression.
- **Vector of Indices**: Executes a for loop of the form:
- for varname=[a b c ...]Specify appropriate values in the **Vector** field in the form a b c ....

Program Flow Control

- **try**
- Execute statements and catch resulting errors
- **Description**
- try marks the beginning of a try-catch statement, a two-part sequence of commands used in detecting and handling errors. The try-catch enables you to bypass default error handling for selected segments of your program code and use your own procedures instead. The two parts of a try-catch statement are a try block and a catch block (see the figure below). The try block begins with the try command and ends just before to the catch command:
- The try block contains one or more commands for which special error handling is required by your program. Any error detected while executing statements in the try block immediately turns program control over to the catch block. Code in the catch block provides error handling that specifically addresses errors that might originate from statements in the preceding try block.
- Both the try and catch blocks may contain additional try-catch statements nested within them.

File I/O

- File input/output in MATLAB is pretty easy
  - Basic ASCII input is really easy if the file is formatted nicely
  - Things get a little more complex if numbers and characters are intermixed
  - MATLAB has built in functions for reading binary files and can handle byteswapping using command options
  - NetCDF and HDF are now built into MATLAB making them quite easy to deal with

ASCII I/O

- There are several ways to read ASCII files into MATLAB
- For a file that is a uniformly formatted array of numbers
  - **load filename** or **load('filename')** or **data = load('filename')** or other variations described in help
  - When used without equals sign, returns array with name of filename without file type extension
- If text is intermixed with numbers an error is returned
- If the number of columns vary per line and error is returned

ASCII I/O

- **data = dlmread(‘filename’, delimiter, R, C)**
  - Variable **data** is the output array
  - Pass file name, text delimiter (i.e. tab, space, comma), starting row and column for reading
  - Delimiter, R, C are optional
  - If no delimiter is specified **dlmread** will infer the delimiter
    - Use ‘’ for treating any and all white space as one delimiter
  - R,C are useful for skipping a header or some type of text information that occurs in the first column or more
    - Note that R and C are zero based in this case

ASCII I/O

- **celldata = textscan(fid, ‘format’, N, …**)
  - **textscan** returns a cell array of text from the file specified by fid using the format line ‘format’. Will perform function call **N** times
  - **fid** is specified by **fid = fopen(‘filename’, permission)**
  - The permission can be one of many choices including read only, write, append, etc
    - **‘r’**, **‘w’**, **‘a’**

ASCII I/O

- The format string is similar in concept to a FORTRAN format string and C conversion characters
- **%Ndvt** is used for integers
- **%N.Dfvt** is used for floating point numbers
- **%Nc** or **%Ns** for a single character or an entire string
- **%Nuvt** for unsigned integers
  - The **vt** in MATLAB is optional and specifies the variable type, e.g. 16-bit, 8-bit. The default is 32-bit without using **vt**
- Examples
  - %9c, %7.2f, %5d32
  - First: read the next nine characters including white space
  - Second: Read the next seven digits (including decimal point) as a floating point number and return 2 places after the decimal point
  - Third: Reads the next five digits as an integer and stores it into a 32-bit integer

ASCII I/O

- File output can be done easily with a couple of functions
- **dlmwrite(‘filename’, M, ‘D’, R, C)**
  - **‘filename’** is the file name as a string, or string variable (without ‘’ in this case)
  - **M** is the data to be written
  - **‘D’** is the delimiter to be used
  - **R**, **C** are the row and column values to skip when starting the write

ASCII I/O

- **fprintf(fid, ‘format’, A)**
  - **fid** is the file identification
  - **‘format’** specifies the format statement
  - **A** is the data to be written
  - **fprintf** is a vector function (talk about that more later in two weeks)
    - Will run through matrix in column order
    - Don’t need loops to output an entire array

ASCII I/O

- Other ways to do input and output
  - **csvread**, **csvwrite**, **fileread**, **textread**
- MATLAB can also directly import Excel or Lotus spreadsheet files

Binary I/O

- Basic binary I/O is done in MATLAB using two commands for reading and writing
- **A = fread(fid, sizeA, precision, skip, machineformat)**
  - **fid** is the file identifier given by **fopen**
  - **sizeA** gives the size of the array **A**. It can be an integer or **[m n]** to specify a vector or array
  - **precision** is the type (int16, float32, etc) of the data being read in
  - **skip** specifies a value of bytes to skip between each read
    - Useful for skipping record headers, e.g. IDL binary files
  - **machineformat** specifies the type of machine the data was written on initially
    - This option corrects for byte-swapping issues

Binary I/O

- **fwrite(fid, A, precision, skip, machineformat)**
  - **fid** specifies the file identification number
  - **A** is the data array to be written in column order
  - **precision** specifies variable type (int8, uint16, float32, etc)
  - **skip** specifies the number of bytes to skip before each write
  - **machineformat** will specify they byte order for writing

NetCDF

- MATLAB has built-in netCDF function calls
- Can read and write netCDF
  - Being familiar with netcdf-3.6.2 C Interface guide will help with the MATLAB commands
- MATLAB can also import/export CDF files
  - Type **help cdfread** or **help cdfinfo** to get more information about CDF in MATLAB

NetCDF

- Basic command naming structure is:

  netcdf.action

  - Where the action could be open, close, abort, getVar, getAtt, inqVarID, etc.
- The syntax of a given command is similar to that of the equivalent netCDF C function call

NetCDF

- **ncid = netcdf.open('filename', 'mode')**
  - Where filename is a string containing the netCDF file name and the mode string is either, 'NC_WRITE', 'NC_SHARE', or 'NC_NOWRITE'
  - This corresponds to read-write access, synchronous file updates and read-only access
- **netcdf.close(ncid)**
- **ncid = netcdf.create('filename', 'mode')**
  - This will create a new netCDF file

NetCDF

- To redefine or create dimensions, variables and attributes you need
  - **netcdf.reDef(ncid)**
  - First, open netCDF file in write mode, then call the redefine function
  - This puts netCDF file into define mode for redefining or adding variables, dimensions and attributes

NetCDF

- **netcdf.inqDim**
  - Return netCDF dimension name and length
- Syntax
  - **[dimname, dimlen] = netcdf.inqDim(ncid,dimid)**
- Description
  - **[dimname, dimlen] = netcdf.inqDim(ncid,dimid)** returns the name, **dimname**, and length, **dimlen**, of the dimension specified by **dimid**. If ndims is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and ndims-1. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.
  - **ncid** is a netCDF file identifier returned by **netcdf.create** or **netcdf.open**.
  - This function corresponds to the **nc_inq_dim** function in the netCDF library C API.

NetCDF

- **netcdf.inqVarID**
- Return ID associated with variable name
- Syntax
  - **varid = netcdf.inqVarID(ncid,varname)**
- Description
  - **varid = netcdf.inqVarID(ncid,varname)** returns **varid**, the ID of a netCDF variable specified by the text string, varname.
  - **ncid** is a netCDF file identifier returned by **netcdf.create** or **netcdf.open**.
  - This function corresponds to the **nc_inq_varid** function in the netCDF library C API.

NetCDF

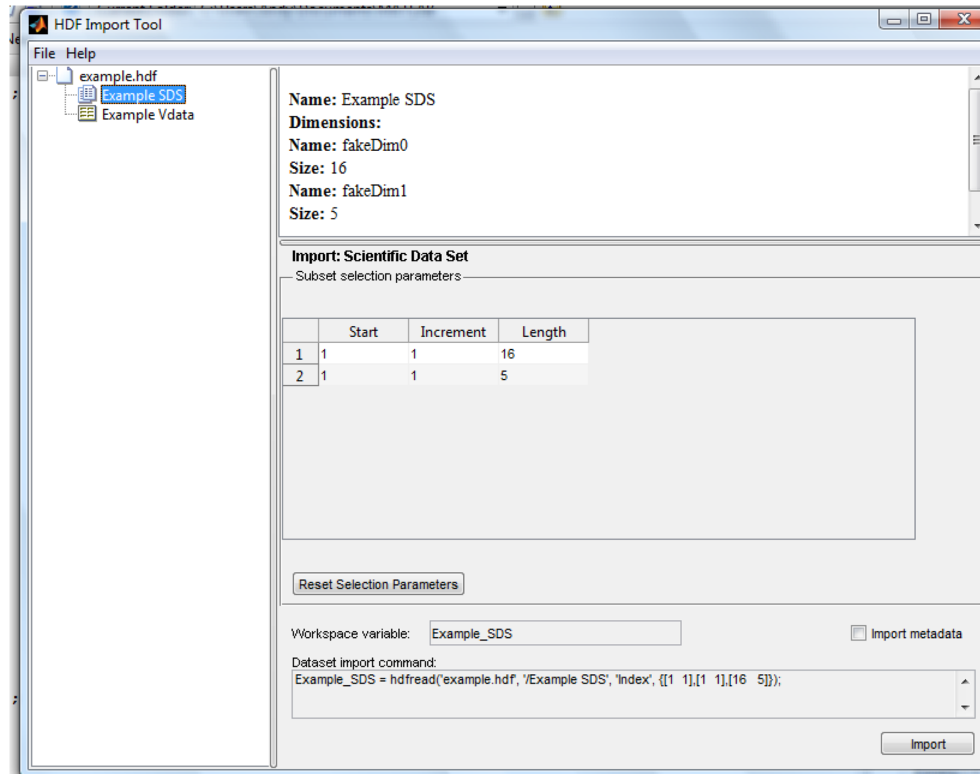
- **netcdf.getVar**
- Return data from netCDF variable
- **Syntax**
  - **data = netcdf.getVar(ncid,varid)**
    data = netcdf.getVar(ncid,varid,start)
    data = netcdf.getVar(ncid,varid,start,count)
    data = netcdf.getVar(ncid,varid,start,count,stride)
    data = netcdf.getVar(...,output_type)
- **Description**
  - **data = netcdf.getVar(ncid,varid)** returns data, the value of the variable specified by **varid**. MATLAB attempts to match the class of the output data to netCDF class of the variable.
  - **ncid** is a netCDF file identifier returned by **netcdf.create** or **netcdf.open.**
  - **data = netcdf.getVar(ncid,varid,start)** returns a single value starting at the specified index, **start**.
  - **data = netcdf.getVar(ncid,varid,start,count)** returns a contiguous section of a variable. **start** specifies the starting point and **count** specifies the amount of data to return.
  - **data = netcdf.getVar(ncid,varid,start,count,stride)** returns a subset of a section of a variable. **start** specifies the starting point, **count** specifies the extent of the section, and **stride** specifies which values to return.
  - **data = netcdf.getVar(...,output_type)** specifies the data type of the return value data. For example, to read in an entire integer variable as double precision, use:
  - **data=netcdf.getVar(ncid,varid,'double');**
  - You can specify any of the following strings for the output data type.
  - ‘int’ ‘double‘ 'int16' 'short‘ 'single‘ 'int8' 'float‘ 'int32‘ 'uint8'
  - This function corresponds to several functions in the netCDF library C API relating to retrieving variables from a netCDF file

HDF

- MATLAB now supports HDF4 and HDF5 formats
- Contains built-in functions and GUI to work with HDF files
  - Scientific datasets (SD), V-data, HDF-EOS data
- **hdftool**
  - Built in GUI that permits you to browse HDF4 or HDF-EOS files

HDF

- Browser interface
  - Can see SDS names, dimensions,
  - VData names, record length
  - Import data into MATLAB



HDF

- **hdfinfo**
  - Returns a structure with information describing HDF file
- **data = hdfread(filename, datasetname)**
  - Pass file name and SDS name, will return all data in dataset
  - **hdfread** can also subset SDS, VData, HDF-EOS data
- **doc hdf** will give you introductory help page for HDF in MATLAB

Questions?