# Armstrong State University
# Engineering Studies
# MATLAB Marina – Structures Primer

**Prerequisites**
The Structures Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, functions, debugging, characters and strings, and cell arrays. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, MATLAB Marina Functions module, MATLAB Marina debugging module, MATLAB Marina Character and Strings module, and MATLAB Marina Cell Arrays module.

**Learning Objectives**
1. Be able to write MATLAB functions and programs that create structures and structure arrays.
2. Be able to write MATLAB constructor functions for structures.
3. Be able to write MATLAB programs and functions that operate on data contained in structure arrays.

**Terms**
structure, field, structure array, constructor, dynamic field, nested structure

**MATLAB Functions, Keywords, and Operators**
struct, . (dot operator), setfield, getfield, isfield, deal, class

**Structures and Structure Arrays**
Structures are collections of data and like cell arrays the data may be of different types. However in structures, the data is organized and accessed by fields rather than an array. Structures consist of a structure name and one or more field names. Structure arrays are arrays where each element in the array is a structure. Structure arrays are commonly used to organize large collections of data such as material or thermodynamic property tables.

**Creating Structures**
Structures can be created by:
- Assigning the value for each field of the structure directly.
- Using the MATLAB `struct` function.
- Using a constructor function (discussed later in the primer).

Examples of creating a structure containing a string and a number are shown in Figures 1a and 1b.

```
>> grocery.item = 'apple';
>> grocery.quantity = 4;
>> grocery
grocery =
    item: 'apple'
    quantity: 4
```

Figure 1a, Structure Creation using Direct Entry

```
>> grocery = struct('item', 'apple', 'quantity', 4);
```

Figure 1b, Structure Creation using `struct` Function

The grocery structure created in Figures 1a and 1b has two fields: item and quantity. The item field holds a string and the quantity field holds a number.

The `struct` function takes pairs of string-cell array arguments corresponding to the field name and field data values for the structure(s) to create and returns the structure(s). The number of structures created is equal to the size of the cell arrays containing the field data values or one if only scalar data values are given. If an empty set is given in place of the arguments, an empty structure is created.

**Accessing/Extracting Data from Structures**

Most MATLAB operations cannot be done directly on data stored in structures. Similar to cell arrays, one typically first extracts the data, operates on it, and moves it back into the structure if desired. The data in the structure's fields can be individually accessed by giving the structure name followed by the dot operator (.) followed by the field name. The structure variable name refers to the entire structure. Figure 2a shows how one would extract the quantity data from a grocery item and then compute the cost of the item assuming apples cost 50 cents each. The extraction of data using the dot operator and field name returns only the data held in that field.

```
>> grocery = struct('item', 'apple', 'quantity', 4);
>> numberApples = grocery.quantity;
>> costApples = numberApples*0.50;
```

Figure 2a, Extracting Data in quantity Field of grocery Structure

Figure 2b shows how one could update data in a single field of a structure, in this example quantity field of the grocery structure. Data can be assigned to a particular field in a structure in a similar manner as creating a structure using direct entry.

```
>> grocery = struct('item', 'apple', 'quantity', 4);
>> numberApples = grocery.quantity;
>> numberApples = numberApples + 2;
>> grocery.quantity = numberApples;
```

Figure 2b, Updating Data in quantity Field of grocery Structure

**Creating Structure Arrays, Constructor Functions**
Structure arrays are an array of structures. Structure arrays can be created by:

- Assigning the value for each field of each structure of the structure array directly.
- Using the MATLAB `struct` function.
- Using a constructor function.
- Empty structures or structure arrays can be created using the MATLAB `struct` function.

Figure 3 shows an example of creating an empty structure and an empty grocery structure using the `struct` function.

```
>> emptyStruct = struct([]);
>> emptyGrocery = struct('item', {}, 'quantity', {});
```

Figure 3, Empty Structures

Figures 4a and 4b, illustrate creating a structure array using direct entry and using the `struct` function.

```
>> grocery(1).item = 'apple';
>> grocery(1).quantity = 4;
>> grocery(2).item = 'orange';
>> grocery(2).quantity = 3;
>> grocery(3).item = 'banana';
>> grocery(3).quantity = 2;
>> grocery
grocery =
1x3 struct array with fields:
    item
    quantity
```

Figure 4a, Structure Array Creation using Direct Entry

```
>> grocery = struct('item', {'apple', 'orange', 'banana'},
'quantity', {4,3,2});
```

Figure 4b, Structure Array Creation using `struct` Function

In Figure 4a, the array indexing selects one of the structures in the structure array and the fields for that structure are accessed using the dot operator. In Figure 4b, the `struct` function returns a structure array since the second argument in each pair of arguments is a cell array rather than a scalar data element.

Rather than directly entering the field values for each element of the structure array as in Figure 4a, the assignment of field values can be assigned to each structure in the structure array using a loop as shown in Figure 5a. A sample run of the code of Figure 5a is shown in Figure 5b.

```
% create structure array of grocery items and quantities
numberItems = input('Enter number of items: ');
% read in items and quantities and save in structure array
for k = 1:1:numberItems
    itemName = input('Enter item: ', 's');
    message = sprintf('Enter quantity of %s: ', itemName);
    itemQuantity = input(message);
    grocery(k).item = itemName;
    grocery(k).quantity = itemQuantity;
end
```

Figure 5a, Structure Array Creation using Direct Entry Enclosed in a Loop

```
Enter number of items: 3
Enter item: apple
Enter quantity of apple: 4
Enter item: orange
Enter quantity of orange: 3
Enter item: banana
Enter quantity of pear: 2

>> disp(grocery)
1x3 struct array with fields:
    item
    quantity
```

Figure 5b, Sample Run and Results of Code of Figure 3a

Populating structure arrays can be very tedious. Typically the data for the structure array is read from the user or a file. One must be careful when creating or modifying structures or structure arrays directly.  A typo on a field name results in the structure having this new unintended field and for a structure array all structures will have this new unintended field. It is generally better to create structures and structure arrays using constructor functions.

Constructor functions are functions specifically designed for creating objects of a data type. The MATLAB `struct` function is a constructor function and one can create their own constructor functions. Constructor functions are particularly useful when creating structure arrays. Constructors typically take one argument for each field in the structure they will create and return a structure or structure array. For example, the constructor `createGrocery` of Figure 6a takes two arguments: an item name and item quantity, and returns a single grocery structure. Optional arguments are used to provide default field values or create a default grocery structure with an empty string for the item name and zero for the item quantity.

```
function result = createGrocery(itemName, itemQuantity)
if (nargin < 1)
    itemName = '';
    itemQuantity = 0;
elseif (nargin < 2)
    itemQuantity = 0;
end
result.item = itemName;
result.quantity = itemQuantity;
end
```

Figure 6a, Constructor `createGrocery`

Figure 6b shows an example of a code segment that uses the constructor of Figure 6a to create a structure array. Notice that the structure of the code in Figure 6a is like that of Figure 5a except rather than creating each structure in the structure array using direct entry, the `createGrocery` constructor is used. An empty structure array is created to store each structure in using the `struct` function and passing in a cell array of empty arrays for the items and an array of zeros for the quantities. This is to prevent the structure array from being resized every time a new structure is created. The `struct` function will create a structure array if the field value parameters are cell arrays.

```
% create structure array of grocery items and quantities
numberItems = input('Enter number of items: ');
% read in items and quantities and save in structure array
grocery = struct('item', cell(1,numberItems), 'quantity',
zeros(1,numberItems));
for k = 1:1:numberItems
    itemName = input('Enter item: ', 's');
    message = sprintf('Enter quantity of %s: ', itemName);
    itemQuantity = input(message);
    grocery(k) = createGrocery(itemName, itemQuantity);
end
```

Figure 6b, Using `createGrocery` Constructor to Create a Structure Array

The built in `struct` function could be used instead of the specialized `createGrocery` as shown in Figure 6c. Advantages of using a specialized constructor function instead of the `struct` function is field names do not have to be specified and the constructor can be tailored to handle default or incomplete field data.

```
% create structure of a grocery item and quantity
% read in item and quantity and save in structure
itemName = input('Enter item: ', 's');
message = sprintf('Enter quantity of %s: ', itemName);
itemQuantity = input(message);
grocery1 = struct('item', itemName, 'quantity', itemQuantity);
grocery2 = creategrocery(itemName, itemQuantity);
```

Figure 6c, Using `struct` and `createGrocery` Constructors to Create Structures

The `createGrocery` constructor could also be designed to create a structure array if a cell array of values were passed in for each field. Figure 7 shows a `createGrocery` constructor function that will create a single structure if one value is passed in for each field or a structure array if cell arrays are passed in for each field.

```
function result = createGrocery(itemName, itemQuantity)
numberItems = length(itemName);
result =
struct('item',cell(1,numberItems),'quantity',cell(1,numberItems));

for k = 1:1:numberItems
    result(k).item = itemName{k};
    result(k).quantity = itemQuantity{k};
end

end
```

Figure 7, Constructor `createGrocery`

Note: the `createGrocery` constructor of Figure 7 assumes that the two cell arrays are the same length and additional code would be needed to provide default values for the shorter of the two if they are not.

**Organizing Data using Structure Arrays**

The fields of a structure may be any data type including arrays, cell arrays, or structures. Generally one should try to keep the data type of the fields as simple as possible so as to avoid complex data organizations where it is difficult to access the data.

Consider organizing the data resulting from evaluating the function $f(t) = 5\cos(10\pi t) - 2\cos\left(20\pi t - \frac{\pi}{2}\right)$ over the time range $0 \le t \le 0.5$ seconds. The program of Figure 8 shows three ways of storing the results of evaluating the function over the time range.

```
% store data as two 1D arrays
t = 0.0:0.01:0.5;
f1 = 5*cos(10*pi*t) - 2*cos(20*pi*t - pi/2);
% store data as a single structure
f2.time = (0.0:0.01:0.5);
f2.values = 5*cos(10*pi*f2.time) - 2*cos(20*pi*f2.time - pi/2);
% store data as 1x51 structure array
numberPoints = 51;
for k = 1:numberPoints
    f3(k).time = 0.0 + (k-1)*0.01;
    f3(k).values  = 5*cos(10*pi*f3(k).time) - ...
        2*cos(20*pi*f3(k).time - pi/2);
end
```

Figure 8, Storing Data Resulting from Evaluating a Function

The variables `t` and f1 store the time and function data using two 1D arrays. The variable `f2` stores both the time and function data using a single structure. The structure `f2` has two fields each of which is a 1D array of numbers. The structure array `f3` stores the data as a 1 by 51 structure array. Each structure in the structure array `f3` has two fields each of which is a single number.

To access the data stored in the structure `f2`, one would first access the field and then index the resulting array stored in the field, for example `f2.time(20)` would return the 20th time value and `f2.values(20)`would return the 20th function value.

To access the data stored in the structure array `f3`, one would first index the structure array and then access the desired field of the structure stored at that place in the structure array, for example `f3(20).time` would return the 20th time value and `f3(20).values` would return the 20th function value.

**Nested Structures**

Nested structures are structures that have a field that is a structure. For example a structure holding information about a person may have a date which itself could be a structure (Figure 9).

Be careful when using nested structures as it can result in complex data types that are difficult to access and use the stored data. To extract the birth month of the `bobInfo` structure, one would use the MATLAB statement `BobsBirthMonth = bobInfo.birthdate.month`.

```
date.day = 20;
date.month = 'July';
date.year = 1995;

bobInfo.birthdate = date;
bobInfo.age = 18;
bobInfo.height = 70;
bobInfo.weight = 180;
```

Figure 9, Nested Structures

**Accessing/Extracting Data in Structure Arrays**
To extract data from a structure array, the structure array is indexed similar to arrays and then the appropriate field(s) are accessed. Indexing a structure array returns a single structure if the index is a scalar or a structure array for an array of indices. Accessing a field of a single structure returns one object whereas access the field of a structure array returns multiple objects. All of the fields of a structure array can be accessed by giving the structure array name followed by the dot operator and field name. This method of accessing the fields returns multiple results so in general this is not that useful the result must be assigned to an equal number of variables.

For example, if `grocery` is a 1 by 5 structure array then `grocery(2)` is the second structure, `grocery(2).item` is the item name of the second structure, `grocery(1:3)`is a 1 by 3 structure array, and `grocery(1:3).item` is three separate item values each of which is a string.

The general rules for accessing/extracting data in structure arrays cell are:

- Array indexing is used to access a single structure or a structure array that is a subset of the original structure array. The indexing operation will result in a single object (single structure or structure array) that can be assigned to a single variable.
- Array indexing using a scalar index followed by the dot operator and fieldname is used to access the contents of a single field in one of the structures. The result is a single object which can be assigned to a single variable.
- Accessing the fields of multiple or a range of structures generally requires a loop. The loop iterates over the indices of the structure array. The structure array is then indexed using the scalar index for that loop iteration followed by the dot operator and fieldname. The result is a single object which is typically stored in an array.

Figure 10b shows the results of indexing the structure array created by the MATLAB code of Figure 10a. Figure 10c shows the results of then accessing the item field of the indexed structure array.

```
% create structure array of grocery items and quantities
itemNames = {'apple', 'orange', 'banana', 'pear', 'lemon',
'grapes'};
itemQuantities = {4, 2, 6, 2, 2, 1};
grocery = struct('item', itemNames, 'quantity', itemQuantities);
```

Figure 10a, `grocery` Structure Array

```
>> grocery(2)
ans =
    item: 'orange'
    quantity: 2
>> grocery(1:3)
ans =
    1x3 struct array with fields:
    item
    quantity
```

Figure 10b, Indexing Results of `grocery` Structure Array

When assigning the result of accessing fields of a structure array, the number of variables on the left hand side of the assignment must equal the number of objects on the right hand side. Notice in Figure 10c, when the field of a structure array is accessed multiple objects are returned and would need to be assigned to the same number of variables, for this example three.

```
>> grocery(2).item
ans =
    orange
>> grocery(1:3).item
ans =
    apple
ans =
    orange
ans =
    banana
```

Figure 10c, Results of Accessing `item` Field of Indexed `grocery` Structure Array

Figure 10d illustrates the different ways of accessing data in structure arrays. In the first line, the structure array is indexed to obtain a single structure. In the second line, the structure array is indexed using an array of indices to obtain a smaller structure array. In the third line, the structure array is indexed and then the item field is accessed to obtain the number of items of a single structure.

```
>> r1 = grocery(2);
>> r2 = grocery(1:3);
>> r3 = grocery(2).item;
>> r4 = grocery(1:3).item;
>> [r5 r6 r7] = grocery(1:3).item;
```

Figure 10d, Accessing Data in `grocery` Structure Array

In the fourth and fifth lines of Figure 10d, the incorrect and correct way to access the same filed of multiple structures is shown. In the fourth line, the structure array is indexed using an array of indices to obtain a smaller structure array. The dot operator is then used to access the item field of each structure in the smaller structure array resulting in three objects. Only one variable is on the left hand side of the assignment, so only the first item field value is saved (the other two item field values are discarded). This is not a syntax error but only one of three access field values is saved. In line 5, the same operation is performed as in line 4 but the left hand side of the assignment has three variables so each of the three returned item field values is stored in the corresponding variable.

**Operating on Data in Structure Arrays**

Most MATLAB operations cannot be done directly on data stored in structures or structure arrays. Similar to cell arrays, one typically first extracts the data, operates on it, and moves it back into the structure or structure array if desired. In general when working with a structure array, a loop is used to iterate through the structure array and the desired field info of each structure is extracted and saved or operated on.

The MATLAB program of Figure 11 shows an example of extracting the quantity information from a grocery structure array and determining the average quantity of the items.

```
% create structure array of grocery items and quantities
itemNames = {'apple', 'orange', 'banana', 'pear', 'lemon', 'grapes'};
itemQuantities = {4, 2, 6, 2, 2, 1};
grocery = struct('item', itemNames, 'quantity', itemQuantities);

% extract the quantity field data
numberItems = length(grocery);
itemQuan = zeros(1,numberItems);
for k = 1:1:numberItems
    itemQuan(k) = grocery(k).quantity;
end

% determine the average quantity
averageQuantity = mean(itemQuan);
```

Figure 11, MATLAB Program to Extract and Operate on Structure Array Data

Using the grocery structure array created by the code segment of Figure 11, the operations shown in Figure 12a are syntactically incorrect although one might think that they would work.

```
>> sum(grocery.quantity)
??? Error using ==> sum
Too many input arguments.

>> sum(grocery(1:end).quantity)
??? Error using ==> sum
Too many input arguments.
```

Figure 12a, Syntax Errors Generated from Incorrectly Operating on Structure Array Data

The `sum` function sums the elements of a vector (or columns of a 2D array). The statements `grocery.quantity` and `grocery(1:end).quantity` return multiple results (in this and sum expects a single argument. Using the `min`, `max`, and `mean` functions on the same data would result in the same kind of syntax error.

The operations shown in Figure 12b are syntactically correct but the result is not what one might expect.

```
>> q(1:length(grocery)) = grocery(1:end).quantity
q =     4     4     4     4     4
>> q = grocery(1:end).quantity
q =    4
```

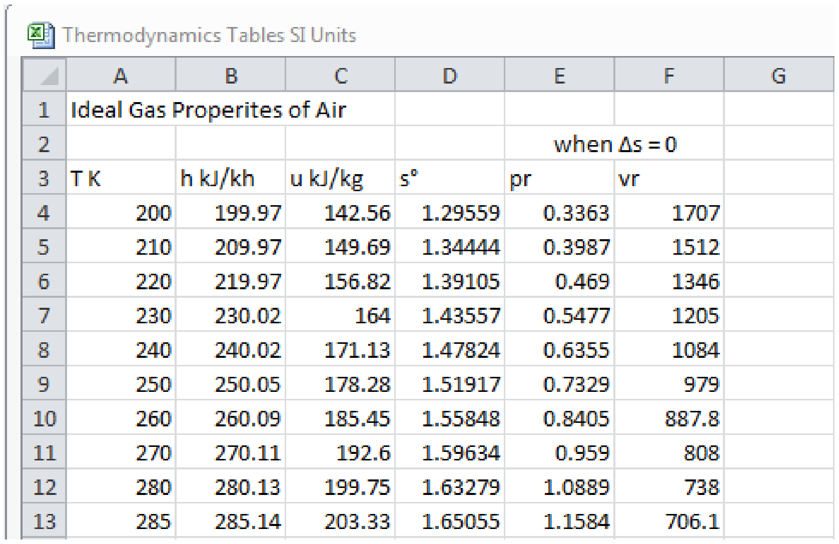
Figure 12b, Syntactically Correct but Logically Incorrect Operations on Structure Array

The statement `grocery(1:end.quantity` returns multiple items, but only a single variable `q` is on the left hand side of the assignment so only the first element extracted from the structure array is assigned. Since the scalar value (only the first element) is being assigned to a 1D array, MATLAB replicates the value according to the size of the 1D array. In general there is no easy way around using a loop to extract the structure array field data one at a time and place them in the appropriate place of an array (usually an array of numbers).

**Properties of Ideal Gas Table Loop Up using a Structure Array**
Figure 13a shows a portion of a thermodynamics table for the ideal gas properties of air. Many engineering problems involve looking up properties from large tables of data that are needed for computations. Figure 13b shows the organization of the structure array containing the data from the ideal gas properties table.

Figures 13d shows the MATLAB function to lookup the ideal gas properties of air given a field name and a value. The function returns a structure containing all six properties (temperature T, enthalpy h, internal energy u, entropy s, relative pressure pr, relative volume vr) for the specified field and value.

Figure 13c shows an alternative function header that returns the properties via separate variables rather than a structure. It would be more awkward to use the function if it returned six variables because even if only one field from the table were desired, the function call would still need up to six variables on the left had side of the assignment, five of which would not be used.



Thermodynamics Tables SI Units

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | Ideal Gas Properites of Air | | | | | | |
| 2 | | | | | when Δs = 0 | | |
| 3 | T K | h kJ/kh | u kJ/kg | s° | pr | vr | |
| 4 | 200 | 199.97 | 142.56 | 1.29559 | 0.3363 | 1707 | |
| 5 | 210 | 209.97 | 149.69 | 1.34444 | 0.3987 | 1512 | |
| 6 | 220 | 219.97 | 156.82 | 1.39105 | 0.469 | 1346 | |
| 7 | 230 | 230.02 | 164 | 1.43557 | 0.5477 | 1205 | |
| 8 | 240 | 240.02 | 171.13 | 1.47824 | 0.6355 | 1084 | |
| 9 | 250 | 250.05 | 178.28 | 1.51917 | 0.7329 | 979 | |
| 10 | 260 | 260.09 | 185.45 | 1.55848 | 0.8405 | 887.8 | |
| 11 | 270 | 270.11 | 192.6 | 1.59634 | 0.959 | 808 | |
| 12 | 280 | 280.13 | 199.75 | 1.63279 | 1.0889 | 738 | |
| 13 | 285 | 285.14 | 203.33 | 1.65055 | 1.1584 | 706.1 | |

Figure 13a, Portion of Thermodynamics Table for Ideal Gas Properties of Air (SI Units)

```
idealGasTableAir =
   1x122 struct array with fields:
    T
    h
    u
    s
    pr
    vr
```

Figure 13b, Organization of Ideal Gas Properties Table Structure Array

```
function [T, h, u, s, pr, vr]  = idealGasTableAirLookup(table,
field, fieldValue)
```

Figure 13c, Alternative Function Header (more awkward to use)

Figure 13e shows three function calls of the `idealGasTableAirLookup` function and what the function returns. The first call has a temperature that is not in the table and the function (and calling program) terminates due to the error handling. The other two function calls show examples of a temperature value that is in the table and a volume that is between two table values.

```
function result = idealGasTableAirLookup(table, fieldName, value)
% if value is outside table boundaries for that field generate an
exception
tableBoundaries = [table(1).(fieldName), table(end).(fieldName)];
if (value < min(tableBoundaries) || value > max(tableBoundaries))
    error('value out of table range');
end
% determine closest table entry corresponding to the value
% start by assuming first table entry is the closest
ind = 1;
closest =  abs(value - table(1).(fieldName));
% check rest of table entries to see if any are closer
for k = 2:1:length(table)
    % compute how close table entry k is to the value
    howClose = abs(value - table(k).(fieldName));
    % update what is thought to be closest table entry
    % if table entry k is closer to the value
    if (howClose < closest)
        ind = k;
        closest = howClose;
    else
        break;
    end
end
% return the table entry
result = table(ind);
end
```

Figure 13d, `idealGasTableAirLookup` Function

In the `idealGasTableAirLookup` function of Figure 13d:

- The vr field data is in the opposite order of the other field data so a conditional statement like `(value < table(1).(field) || value > table(end).(field))` would not detect a vr value that was outside the table entry boundaries. Instead the minimum and maximum values in the field are determined and compared to the passed in value. It is typically considered poor practice to terminate programs when errors are detected. In the `idealGasTableLookup` function, if the desired value is outside the bounds of the table for that field, it does not make sense to do any further work with the table values. Alternatively, a try-catch block could be added to the function to deal with the error or the function could return the lowest or highest table entry along with an indication that the table values are probably not meaningful for that field value.
- The best matched table entry is found by comparing the table entry field data of each element in the structure array to the value and determining which entry is closest to the value. If the table entry being compared is closer than the previously compared table entry,

the index of what is considered the closest table entry and how close the values are is updated.

- The T, h, u, s, and pr field data are in ascending order and the vr field data is in descending order. The search for the closest table entry can be stopped (accomplished with the `break` statement in `else` part of conditional statement inside the `for` loop)  once the entry being checked is not closer than the previous checked entry as any subsequent entry will be further away and be a worse match. Normally the use of `break` to terminate loops is discouraged. In this case, when searching for values in an ordered table, once the desired value is found there is no sense in searching remainder of table and if the table is large this can save substantial processing time.

```
>> r = idealGasTableAirLookup(idealGasTableAir,'T',100)
Error using idealGasTableAirLookup (line 5)
value out of table range

>> r = idealGasTableAirLookup(idealGasTableAir,'T',400)
r =
     T: 400
     h: 400.9800
     u: 286.1600
     s: 1.9919
    pr: 3.8060
    vr: 301.6000

>> r = idealGasTableAirLookup(idealGasTableAir,'vr',489)
r =
     T: 330
     h: 330.3400
     u: 235.6100
     s: 1.7978
    pr: 1.9352
    vr: 489.4000
```

Figure 13e, Sample Function Calls of `idealGasTableAirLookup` Function

Figure 13f shows an alternative implementation of the `idealGasTableAirLookup` function using array and element by element operations instead of a loop. Typcailly when extracting data from a structure array a loop is used. In this version of the `idealGasTableAirLookup` function, the field data for one field is extracted using the statement `fieldData = [table(1:1:end).(fieldName)];`. Recall that when accessing the fields of multiple structures, multiple objects are returned. In this case , since all the fields contain the same data type (real numbers) the multiple return objects can be concatenated into a 1D array avoiding the typical assignment of multiple object problem with extracting field data from structure arrays. This method of extracting structure array field data does not always result in a nice array, but can be used when the extracted data consists of only numbers of the same data type.

```
function result = idealGasTableAirLookup(table, fieldName, value)
% if value is outside table boundaries for that field generate an
exception
tableBoundaries = [table(1).(fieldName), table(end).(fieldName)];
if (value < min(tableBoundaries) || value > max(tableBoundaries))
    error('value out of table range');
end
% extract field data from table structure array
fieldData = [table(1:1:end).(fieldName)];
% determine how close each table entry is to the value
howClose = abs(value - fieldData);
% determine the structure array index of the closest table entry
[smallestHowClose, ind] = min(howClose);
% return the table entry
result = table(ind);
end
```

Figure 13f, `idealGasTableAirLookup` Function Implemented using Array Operations

**Additional Useful Functions for Operating on Structures**

The dot operator and `struct` and `length` functions are commonly used for operating on structures and structure arrays. The table of Figure 14 gives some additional built in functions that may be useful when working with structures and structure arrays. MATLAB's documentation has more details on these and examples of their use.

| Function | Description |
|---|---|
| `ISSTRUCT(S)` | Returns logical true (1) if S is a structure and logical false (0) otherwise. |
| `ISFIELD(S,FIELD)` | Returns true if the string FIELD is the name of afield in the structure array S. |
| `F = GETFIELD(S,'field')` | Returns the contents of the specified field. This is equivalent to the syntax F = S.field. S must be a 1-by-1 structure. |
| `S = SETFIELD(S,'field',V)` | Sets the contents of the specified field to the value V. This is equivalent to the syntax S.field = V. S must be a 1-by-1 structure. The changed structure is returned. |
| `S = RMFIELD(S,'field')` | Removes the specified field from the m x n structure array S. The size of input S is preserved.<br>S = RMFIELD(S,FIELDS) removes more than one field at a time when FIELDS is a character array or cell array of strings. The changed structure is returned. The size of input S is preserved. |
| `NAMES = FIELDNAMES(S)` | Returns a cell array of strings containing the structure field names associated with the structure s. |
| `S = CELL2STRUCT(C,FIELDS,DIM)` | Converts the cell array C into the structure S by folding the dimension DIM of C into fields of S. SIZE(C,DIM) must match the number of field names in FIELDS. FIELDS can be a character array or a cell array of strings. |
| `C = STRUCT2CELL(S)` | Converts the M-by-N structure S (with P fields) into a P-by-M-by-N cell array C. |
| `S = class(OBJ)` | Returns the class name of an object, for example double or struct. |

Figure 14, Useful Operators and Functions for Operating on Structures

Last modified Thursday, November 13, 2014