Lecture 16
MATLAB III: More Arrays and Design Recipe

Last Time (lectures 14 & 15)

Lecture 14: MATLAB I

- "Official" Supported Version in CS4: MATLAB 2018a
- How to start using MATLAB:
  - CS Dept. Machines - run '`cs4_matlab`'
  - Total Academic Handout (TAH) Local Install - software.brown.edu
  - MATLAB Online (currently 2019a) - matlab.mathworks.com
- Navigating the Workspace (command window, variables, etc.)
- Data types in MATLAB (everything is a 64-bit double float by default!)
- MATLAB Programs
  - scripts (like Python)
  - functions (file-based, outputs defined in signature)
- Anonymous functions and overwriting function names (oops!)

Last Time (lectures 14 & 15)

Lecture 15: MATLAB II

- Conditional Statements
  - if...end
  - if...else...end
  - if...elseif...else...end
  - switch...end
- Arrays and Matrices (default numeric type)
  - scalars (1x1 value)
  - 1D vectors (1xN or Nx1 arrays)
  - 2D matrices (MxN)
  - `linspace(a, b, n)` vs. `first:step:max`
- Array concatenation, slicing, and indexing
- Array Manipulation
  - zero-padding
  - removing elements
  - row-to-column `x(:)`
- Size of arrays (`numel` and `size`; **not** `length`)

Lecture 16 Goals: MATLAB III

- Multi-dimensional arrays:
  - Applying built-in functions to matrices
  - Scalar operations on matrices
  - Element-wise operations on matrices
  - Logical array comparisons
  - Array indexing with ‘find’
  - 3D arrays

Arrays as function arguments

- Many MATLAB functions that work on single numbers will also work on entire arrays; this is very powerful!
- Results have the same dimensions as the input, results are produced "elementwise"
- For example:

```
>> av = abs([-3 0  5 1])
av =
     3     0     5     1
```

Powerful Array Functions

- There are a number of very useful function that are built-in to perform operations on vectors, or column-wise on matrices:
  - **min** the minimum value
  - **max** the maximum value
  - **sum** the sum of the elements
  - **prod** the product of the elements
  - **cumprod** cumulative product
  - **cumsum** cumulative sum

min, max Examples

```
>> vec = [4  -2  5  11];
>> min(vec)
ans =
    -2
>> mat = randi([1, 10], 2,4)
mat =
     6     5     7     4
     3     7     4    10
>> max(mat)
ans =
     6     7     7    10
```

- Note: the result is a scalar when the argument is a vector; the result is a *1 x n* vector when the argument is an *m x n* matrix

sum, cumsum vector Examples

- The **sum** function returns the sum of all elements; the **cumsum** function shows the running sum as it iterates through the elements (4, then 4+-2, then 4-2+5, and finally 4-2+5+11)

```
>> vec = [4  -2  5  11];
>> sum(vec)
ans =
    18
>> cumsum(vec)
ans =
     4     2     7    18
```

What is the value of b?

```
a = [2 3 1; -2 0 -6; 8 7 -1];
b = min(a);
```

What is the value of b?

A) -6

B) [-2 0 -6]

C) [1 -6 -1]

D) [-6 -6 -6]

What is the value of b?

```
a = [2 3 1; -2 0 -6; 8 7 -1];
b = min(a);
```

What is the value of b?

A) -6

B) [-2 0 -6]

C) [1 -6 -1]

D) [-6 -6 -6]

What is the value of b?

```
a = [2 3 1; -2 0 -6; 8 7 -1];
b = min(a');
```

What is the value of b?

A) -6

B) [-2 0 -6]

C) [1 -6 -1]

D) [-6 -6 -6]

What is the value of b?

```
a = [2 3 1; -2 0 -6; 8 7 -1];
b = min(a');
```

What is the value of b?

A) -6 **B) [-2 0 -6]**

C) [1 -6 -1] **D) [-6 -6 -6]**

What is the value of b?

```
a = [2 3 1; -2 0 -6; 8 7 -1];
b = min(a(:));
```

What is the value of b?

A) -6

B) [-2 0 -6]

C) [1 -6 -1]

D) [-6 -6 -6]

What is the value of b?

```
a = [2 3 1; -2 0 -6; 8 7 -1];
b = min(a(:));
```

What is the value of b?

A) -6

B) [-2 0 -6]

C) [1 -6 -1]

D) [-6 -6 -6]

sum, cumsum matrix Examples

- For matrices, most functions operate column-wise:

```
>> mat = randi([1, 10], 2,4)
mat =
     1    10     1     4
     9     8     3     7
>> sum(mat)
ans =
    10    18     4    11
>> cumsum(mat)
ans =
     1    10     1     4
    10    18     4    11
```

The **sum** is the sum for each column; **cumsum** shows the cumulative sums as it iterates through the rows

prod, cumprod Examples

- These functions have the same format as **sum/cumsum**, but calculate products

```
>> v = [2:4   10]
v =
     2     3     4    10
>> cumprod(v)
ans =
     2     6    24   240
>> mat = randi([1, 10], 2,4)
mat =
     2     2     5     8
     8     7     8    10
>> prod(mat)
ans =
    16    14    40    80
```

Overall functions on matrices

- When functions operate column-wise for matrices, make nested calls to get the function result over all elements of a matrix, e.g.:

```
>> mat = randi([1, 10], 2,4)
mat =
     9     7     1     6
     4     2     8     5
>> min(mat)
ans =
     4     2     1     5
>> min(min(mat))
ans =
     1
```

Overall functions on arrays

- Alternatively, since linear indexing arranges all the elements of an array into a column, you can also use this approach.

```
>> m = max(A(:)) % Find max of A, regardless of dim.
```

Scalar operations

- Numerical operations can be performed on every element in an array
- For example, ***Scalar multiplication:*** multiply every element by a scalar

```
>> [4  0  11] *  3
ans =
    12     0    33
```

- Another example: scalar addition; add a scalar to every element

```
>> zeros(1,3) + 5
ans =
     5     5     5
```

Array Operations

- ***Array operations*** on two matrices A and B:
  - these are applied between individual elements
  - this means the arrays must have the same dimensions
  - In MATLAB:
    - matrix addition: A + B
    - matrix subtraction: A – B   or   B – A
  - For operations that are based on multiplication (multiplication, division, and exponentiation), a dot must be placed in front of the operator. Unless you're doing linear algebra, this point-wise approach is generally what you want.
    - array multiplication: A .* B
    - array division: A ./ B, A .\ B
    - array exponentiation A .^ 2
  - matrix multiplication: A*B is NOT an element-wise operation

Logical Vectors and Indexing

- Using relational and logical operators on a vector or matrix results in a **logical** vector or matrix

```
>> vec = [44  3  2  9  11  6];
>> logv = vec > 6
logv =
1    0    0    1    1    0
```

- Can use this to index into a vector or matrix, index and matrix dimensions must agree (logical linear indexing also OK)

```
>> vec(logv)
ans =
44    9   11
```

Element-wise logical operators

- | and & applied to arrays operate elementwise; i.e. go through element-by-element and return logical 1 or 0

```
>> [1 2 3 -1 1]>[0 1 2 1 0]
ans =  1×5 logical array
   1   1   1   0   1
```

- || and && are used for scalars

True/False

- **false** equivalent to logical(0)
- **true** equivalent to logical(1)

- **false(m,n)** and **true(m,n)** create matrices of all **false** or **true** values

Logical Built-in Functions

- **any**, works column-wise, returns true for a column, if it contains any true values
- **all,** works column-wise, returns true for a column, if all the values in the column are true

```
>> M = randi([-5 100], m, n)
>> any(M<0 | M==5) % returns a 1 x n vector
                   % elements are true if corresponding
                   % column in M has any negative
                   % entries or any 5s in it.
>> all(M(:)>0) % true if all elements strictly positive
```

Finding elements

- **find** finds locations and returns indices

```
>> vec
vec =
    44      3      2      9     11      6
>> find(vec>6)
ans =
     1      4      5
```

- **find** also works on higher dimensional arrays

```
[i,j] = find(M>0) % returns non-zero matrix
  indices
ind = find(A>0) % returns linear array indices
```

Comparing Arrays

- The **isequal** function compares two arrays, and returns logical **true** if they are equal (all corresponding elements) or **false** if not

```
>> v1 = 1:4;
>> v2 = [1 0 3 4];
>> isequal(v1,v2)
ans =
     0
>> v1 == v2
ans =
     1     0     1     1
>> all(v1 == v2)
ans =
     0
```

3D Matrices

- A three dimensional matrix has dimensions *m x n x p*
- Can create with built-in functions, e.g. the following creates a *3 x 5 x 2* matrix of random integers; there are 2 layers, each of which is a *3 x 5* matrix

```
>> randi([0 50], 3,5,2)
ans(:,:,1) =
    36    34     6    17    38
    38    33    25    29    13
    14     8    48    11    25
ans(:,:,2) =
    35    27    13    41    17
    45     7    42    12    10
    48     7    12    47    12
```

Functions diff and meshgrid

- **diff** returns the differences between consecutive elements in a vector
- **meshgrid** receives as input arguments two vectors, and returns as output arguments two matrices that specify separately x and y values

```
>> [x y] = meshgrid(1:3,1:2)
x =
     1     2     3
     1     2     3
y =
     1     1     1
     2     2     2
```

Where could meshgrid be useful?

Common Pitfalls

- Attempting to create a matrix that does not have the same number of values in each row
- Confusing matrix multiplication and array multiplication. Array operations, including multiplication, division, and exponentiation, are performed term by term (so the arrays must have the same size); the operators are .*, ./, .\, and .^.
- Attempting to use an array of **double** 1s and 0s to index into an array (must be **logical**, instead)
- Attempting to use || or && with arrays. Always use | and & when working with arrays; || and && are only used with logical scalars.

Programming Style Guidelines

- Extending vectors or matrices is not very fast, avoid doing this too much
- To be general, avoid assuming fixed dimensions for vectors , matrices or arrays. Instead, use **end** and **colon :** in context, or use **size** and **numel**

```
>> len = numel(vec);
>> [r, c] = size(mat);
>> last_col = mat(:, end);
```

- Use **true** instead of **logical(1)** and **false** instead of **logical(0)**, especially when creating vectors or matrices.

DESIGN Recipe

Testing

- Even simple functions can be deceptively hard to verify as correct just by “looking at them”
- However, it is easy to test functions on data you understand (and know what the correct answer should be)
- As functions and programs (which may use lots of functions) get more complicated this becomes very important

assert

In MATLAB, the **`assert`** function allows one to easily perform a test

`assert(expr, message)`

Stops execution and prints our the message when expr evaluates to false.

Examples

- test_triArea.m
- test_myQuadRoots.m

Testing is Programming

- We’ve discovered developing tests first (before writing any functions) often speeds the development process and helps ensure programs work correctly
- In fact, designing tests should be viewed as a part of programming even though you aren’t actively coding a solution.

Design Recipe

Design Recipe

1. Develop important Test Cases – (actually code them, requires you to first create function header)
2. Code function body
3. Test!
4. Fix code, re-Test until working correctly

Example: myFtoC

- Use the Design Recipe to solve the following problem:

“Write a function converts degrees Fahrenheit to degrees Celsius.”

Example: myFtoC

1. Write test_myFtoC
2. Write myFtoC
3. Run test_myFtoC
4. Fix code, re-test until working correctly
5. Look at code, identify any pertinent additional tests
6. Retest, until working correctly

Done!

Example: myFtoC

- test_myFtoC.m
- myFtoC.m

Example: quadMin

- Use the Design Recipe to solve the following problem:

“Write a function that finds x that minimizes

ax^2+bx+c

in the interval [L,R]. Assume a>=0, L<R.”

Example: quadMin

"Write a function that finds x that minimizes

ax^2+bx+c

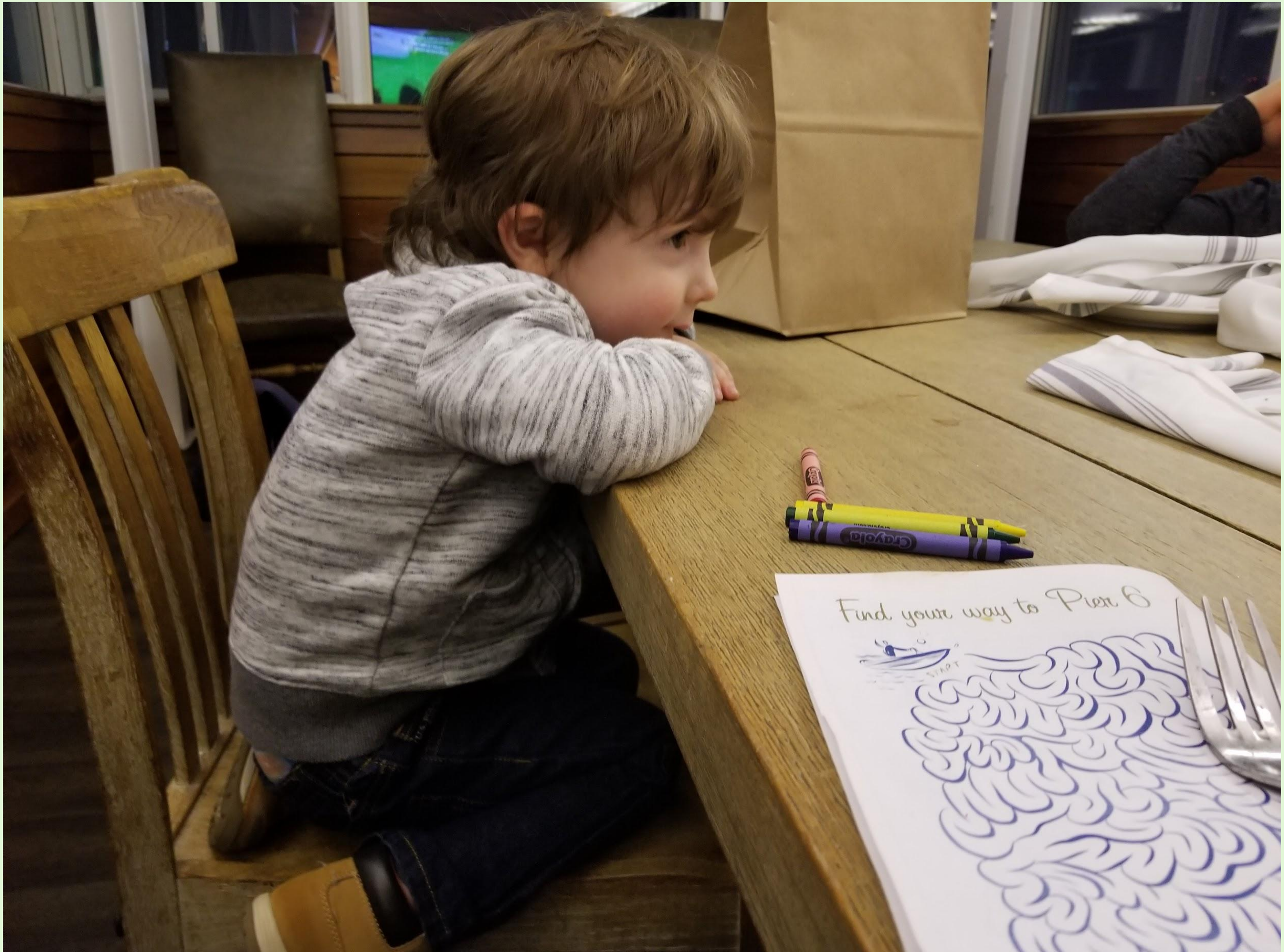
in the interval [L,R]. Assume $a>=0$, $L<R$."

What kind of tests should we have?

What are the cases?

Example: quadMin

- test_quadMin.m


Crayola
Find your way to Pier 6
START