Why is MATLAB so great?

- Excellent matrix/vector handling
- Easy visualizations
- Easy to check variable values
- Easy debugging
- Fast to write programs
- Profiler to see what parts of a program are most time-consuming
- Object-oriented

Nobody’s perfect, not even MATLAB

- Programming environment, not an actual programming language
- Expensive $$$$$$
- May not be ubiquitous in certain fields
- Computationally slow
- Can be difficult to incorporate outside libraries, e.g., Gurobi and CPLEX
Programming in MATLAB

- Mostly the same as in C and Java
- Some slight syntax differences in control structures
- Manipulation of matrices and vectors is far more advanced than in C and Java

Major differences

- No variable types, and no need to declare variables!
  - If you want a variable, just start using it.
  - This flexibility is both good and bad.
- Array indices start at 1, not 0
- There are no “programs”; only a series of functions or individual statements

Two ways to code in MATLAB

- Command window: Write individual statements
- Editor: Write functions or individual statements
Now for the basic statement syntax

MATLAB files are text files with extension .m
- Just like Java files are text files with extension .java
- But, you can evaluate individual highlighted lines of code using shortcuts!\(^1\)
  - Windows: F5
  - Mac: shift+F7 (shift+fn+F7 on a laptop)

If it’s not mentioned, it’s because it is identical to Java.

\(^1\)Different shortcuts for different OSs and versions of MATLAB; best to check your own version/OS.

---

if statements

```matlab
if x < 10
    ...
elseif x >= 15
    ...
elseif x ~= 12
    ...
else
    ...
end
```

while loops

- No do-while loops
- We will do for loops after learning about vectors

```matlab
while x < 10
    ...
end
```
Vectors

A vector is just a series of numbers, as a column or a row (just like math)

```matlab
% row vector
row = [1 2 3];
% column vector
col = [4; 5; 6];
% row vector 2 through 6: [2 3 4 5 6]
x = 2:6;
% row vector of 3 through 13 in steps of 2: [3 5 7 9 11]
y = 3:2:11;
% concatenate row vectors: [2 3 4 5 6 3 5 7 9 11]
z = [x y];
% transpose a vector
row = row';
% concatenate column vectors: [1 2 3 4 5 6]
z2 = [row; col];
```

Accessing vectors

```matlab
% row vector 1 through 6: [1 3 5 7 9 11]
x = 1:2:12
x(3) % 5
x(6) % 7
x(end) % x(6) = 11
oddInds = [1 3 5];
x(oddInds) % [1 5 9]
x([2 3 6]) % [3 5 11]
x([6 3 2]) % [11 3 5]
length(x) % 6
size(x) % 1 x 6 (row by col)
% inner (dot) product (row*col = scalar)
detprod = x * x';
% norm(x) % vector norm (2-norm; sqrt(sum of x_i^2))
sum(x) % sum of x
mean(x) % mean of x
min(x) % min of x
max(x) % max of x
std(x) % standard deviation of x
```

for loops

```matlab
% a pre-defined vector
x = 2:6;
% loop through values of x
for ii = x
    ii
end
% loop through x using numbered indices
for ii = 1:length(x)
    fprintf("%d. %d\n", ii, x(ii))
end
```
Matrices of any dimension

Very similar to vectors (actually, vectors are just 1D matrices)

```matlab
% A 3x3 matrix with some values
A = [8 0 0; 6 6 0; 3 4 8]

zeros(5,3) % 5x3 matrix of zeros
ones(2,6,4) % 2x6x4 matrix of ones
rand(4,3) % 4x3 matrix of random numbers in [0,1]
eye(4) % 4x4 identity matrix
magic(3) % 3x3 magic matrix (all row sums = col sums)
Ainv = A^-1 % matrix inverse
chol(A) % Cholesky factorization
A(3,1) % 3
A(2,:) % row 2, all columns: [6 6 0]
A(:,3) % column 3, all rows: [0 0 8]
A(1:2,2:3) % sub-matrix of A (top right corner)
sum(A) % sum of all columns
sum(sum(A)) % sum of all values
min(A') % min of all rows
min(min(A')) % min of all values
```

```
src/MATLAB/matrices.m
```

---

Plotting: Super easy

```matlab
y1 = 0:2:20;
plot(y1) % opens a new figure and plots y1

% simple enhancements
grid on
xlabel('index')
ylabel('y values')
title('A sample plot')

% a new plot; same y values, x values start at 0
x1 = 0:length(y1) - 1;
plot(x1, y1) % overwrites previous figure!
figure % create a new figure, no overwriting existing figures
hold all % put future plots in same figure with new color
y2 = x1.^2; % element-wise ^2 operation
plot(x1, y1)
plot(x1, y2)
scatter(x1, y1) % scatter plot
strLegendArray = {'y1 curve'} {'y2 curve'} {'y1 points'};
legend(strLegendArray)
```

```
src/MATLAB/plotting.m
```

---

Plotting options

- Line color, line width, line style, font sizes, etc.
- Look in MATLAB help for details
Functions

- Like Java classes, each MATLAB function should be in its own file named with the function name.
- No pass by reference, but can return any number of values!
- No `return` statements; just assign the output variables their values.
- Function files should be saved in the MATLAB path or in the directory where they will be used.
- If you put comments immediately after the function declaration, typing "help myfun" will print those comments!

A function example

```matlab
function [mymin, mymax, mymean, mystd, mymed] = getStats(x)

% Gets the following statistics on 1D vector x:
% minimum, maximum, mean, standard deviation, median
% NEW COMMENTS!
mymin = min(x);
mymax = max(x);
mymean = mean(x);
mystd = std(x);

% let's do the median by hand, though we could use median(x)
sorted = sort(x);
midpt = ceil(length(x)/2);
if mod(length(x),2) == 0
    mymed = (sorted(midpt) + sorted(midpt+1))/2;
else
    mymed = sorted(midpt);
end
```

MATLAB even does optimization!

- `linprog`: Linear programming
- `quadprog`: Quadratic programming
- `bintprog`: Binary integer programming
- `fmincon`: Generalized minimizer for nonlinear objective function

\(^2\text{Requires optimization toolbox}\)
Linear programming with linprog

\[ [x, z] = \text{linprog}(c, A, b, Aeq, beq, lb, ub) \]

minimize \[ z = c^T x \]
subject to
\[ Ax \leq b \]
\[ A_{eq}x = b_{eq} \]
\[ lb \leq x \leq ub \]

LP example

minimize \[ z = -5x_1 - 4x_2 - 6x_3 \]
subject to
\[ x_1 - x_2 + x_3 \leq 20 \]
\[ 3x_1 + 2x_2 + 4x_3 \leq 42 \]
\[ 3x_1 + 2x_2 \leq 30 \]
\[ x_1 + x_2 + x_3 = 18 \]
\[ 0 \leq x_1 \leq 5 \]
\[ 0 \leq x_2 \leq 15 \]
\[ 0 \leq x_3 \leq 6 \]

\% The problem parameters
\[ c = [-5, -4, -6] \]
\[ A = [1 -1 1; 3 2 4; 3 2 0] \]
\[ b = [20, 42, 30] \]
\[ A_{eq} = [1 1 1] \]
\[ b_{eq} = [18] \]

\% bounds on decision variables
\[ lb = \text{zeros}(3,1) \]
\[ ub = [5; 15; 6] \]

% linear program optimization
\[ [x, z] = \text{linprog}(c, A, b, A_{eq}, b_{eq}, lb, ub) \]

LP variations

maximize \[ z = x_1 + 2x_2 \]
subject to
\[ x_1 + 3x_2 \leq 16 \]
\[ x_1 + x_2 \leq 7 \]
\[ x_1, x_2 \geq 0 \]

\% ApophisPharm problem
\[ c = [1 2] \]
\[ A = [1 3; 1 1; -1 0; 0 -1] \]
\[ b = [16; 7; 0; 0] \]

% linear program optimization; leave out unused parameters
\[ [x, z] = \text{linprog}(c, A, b) \]

% linprog ALWAYS does minimization!
\[ [x, z] = \text{linprog}(-c, A, b) \]
\[ z = -z \]

% Different non-negativity handling
\[ A = [1 3; 1 1] \]
\[ b = [16; 7] \]
\[ lb = \text{zeros}(2,1) \]
\[ [x, z] = \text{linprog}(c, A, b, [], [], lb) \]
A really useful example program

- What’s something that we do as IEs that requires a lot of matrix math?
  ...
  ...
  ...

- The (revised) simplex method!

Some assumptions

- We’ll assume that the problem is a minimization with all equality constraints.

- The user will provide the starting basis\(^3\).

\(^3\)We could figure this out on our own if we really wanted to ... how?

The simplex method

- Initial BFS: Get from user
- Solution exists: Yes if \(\min(B^{-1}b) \geq 0\)
- Solve: \(x_B = B^{-1}b, z = c_B^\top x_B\)
- Entering variable exists: Yes if \(\max(c_B^\top B^{-1}N - c_N^\top) > 0\)
- Departing variable exists: Yes if \(\min_j \{B^{-1}\mathbf{a}_j\} > 0, \{B^{-1}\mathbf{b}_j / (B^{-1}\mathbf{a}_j)\} \geq 0\)
Pseudo-code

Simplex method

Require: A, b, c, B (basis set) from user with all slack variables included
1: if min(B⁻¹b) < 0 then
2: return “Infeasible solution”
3: end if
4: while true do
5: if max(cᵀB⁻¹N − cᵀN) > 0 then
6: Entering variable e = argmax(cᵀB⁻¹N − cᵀN)
7: else
8: return “Optimal solution”, [x, z]
9: end if
10: if min_j(B⁻¹aₑ_j) > 0 then
11: Departing variable d = argmin_j(B⁻¹b_j / (B⁻¹aₑ_j)) ≥ 0 then
12: end if
13: return “Unbounded solution”
14: end if
15: Update basis: B = B \ d \cup e
16: end while

Streamlining code

- Lots of the simplex calculations are repetitive, so we should write a function to calculate all the values of interest:

\[ [z₀, zCoeffs, BinvA, xb] = getSimplexValues(A, b, c, BB) \]

- Although it seems like finding the entering and departing variables is complicated, we can actually do it in just 1-2 lines in MATLAB

Write the code

- We’ll do it together.

- I suggest starting with the getSimplexValues function.
Fancy simplex program

Prof. Aleman’s personal simplex program—which includes graphs, tableau printouts, and the Two-Phase Method—is in src/MATLAB/simplex if you want to use it to help check your work in other courses.

IEs live for efficiency

- Save time by not re-typing equations in the command window: Just hit ↑ and ↓ to see your previous commands. You can even start typing the first few letters and then hit ↑ and ↓.

- Debugging: If your program crashes, type "dbstop if error" into the command window. The next time you a run program that crashes, MATLAB will stop at the point of the crash and you can play with all the variables and equations to see what is wrong. Use "dbquit" to quit the debugger.

- Save your work: You can save variables and load them again later:
  
  ```matlab
  save('myvars.mat', 'A', 'b', 'c', 'BB')
  load('myvars.mat')
  ```