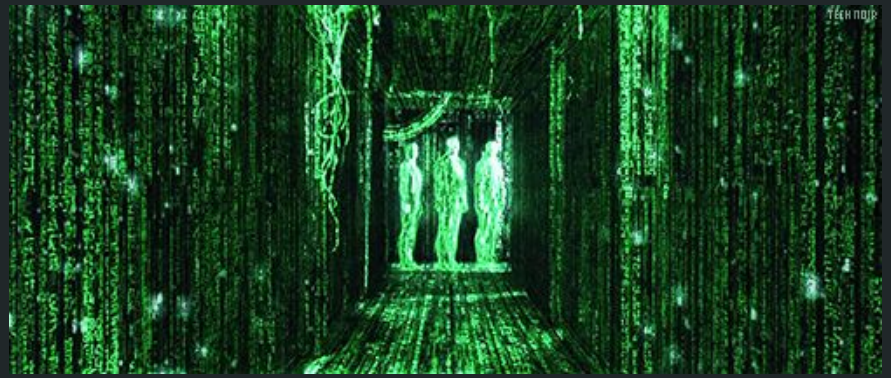

matrix monday!

bit.ly/clps950_2pm

quick sidenote about functions + homework

```
function [x3] = add_numbers(x1,x2)
% function must sum together numbers. for example,
% if x1=5 and x2=3, should return 8.
x1 = 5; x2 = 3;
x3 = x1 + x2;
end
```

what's wrong with this function? remember recipe metaphor from week 1!

Matrices

everything you've been learning about with arrays / vectors applies now in matrices. **arrays are just one dimensional matrices**.

now we will use the full power of what we discussed last week – we can make much bigger data storage structures to organize information!

we're going to go through a lot of functions today – you will get them all reinforced in the tutorials. you will learn them as you use them and through trial and error! you can do this!

Matrices

everything you've been learning about with arrays / vectors applies now in matrices. **arrays are just one dimensional matrices**.

now we will use the full power of what we discussed last week – we can make much bigger data storage structures to organize information!

example for today: scores in a class

imagine you teach a class of 5 students and have grades for three homework assignments and a test for each of 2 semesters, plus a final. so you have 9 assignments and 5 students. we can make this into a 5x9 matrix ***grades*** *where* ***rows represent the students*** *and the* ***columns represent the assignments***

if you want to follow along: `grades = 3*randn(5,9) + 90`

Storing data in matrices

matrices work for adding and accessing data the same way as arrays – we just need to now be explicit about exactly what *dimension* we're working with.

for example, if I said **grades(3)** should matlab return the 3rd student or the 3rd assignment? and which value of those?

so, we need to pass more specific information to make sure we get what we want. rember, just like you saw in the subplots you made, matlab always uses **row, column** for indexing. to get the 3rd student's second assignment, I would call **grades(3,2)**

with matrices we can use the : colon operator if we want **everything** from a certain row or column. so **grades(3,:)** will give us a 1x9 array of assignments for the 3rd student. what will **grades(:,end)** give me? what about **grades(1:2:5, 3:-1:2)?**

Creating matrices

in the example I used a function **randn** we already learned about, but I might also want to type values by hand to make a matrix. we can use semicolons inside of square brackets to tell matlab to make a new **row** in our matrix for the data.

this will only work for well formed information! why didn't the second command work?

```
>> g = [99, 95, 93; 88, 45, 78]

g =

    99    95    93
    88    45    78

>> g = [99, 95, 93; 88, 45]
Error using vertcat
Dimensions of arrays being concatenated are not consistent.

>>
```

Creating matrices

just like we could turn row vectors into column vectors, we can turn the rows and columns in our matrices into each other the same way! imagine I have two students and 3 assignments, but I want rows to be assignments. I can use the apostrophe to flip them

```
>> g = [99, 95, 93; 88, 45, 78]

g =

    99    95    93
    88    45    78

>> g'

ans =

    99    88
    95    45
    93    78
```

Creating matrices

we can also make matrices by putting together multiple vectors or even other matrices! as long as the sizes make sense to matlab to glue together, it will try to do it. to see this, let's first take the **mean** of the grades on each test. if this is the documentation for mean, should I call **mean(grades,1) or mean(grades,2) ?**

```
>> help mean
 mean   Average or mean value.
    S = mean(X) is the mean value of the elements in X if X is a vector.
    For matrices, S is a row vector containing the mean value of each
    column.
    For N-D arrays, S is the mean value of the elements along the first
    array dimension whose size does not equal 1.

    mean(X,'all') is the mean of all elements in X.

    mean(X,DIM) takes the mean along the dimension DIM of X.
```

Creating matrices

we want to compare the grades from the class we're teaching and what we did in the past. so with means over all students in each, we can make a new matrix which has rows for semesters and columns still assignments

avg_s22 = mean(grades, 1);

avg_s21 = mean(oldgrades,1);

both_sem_grades = [avg_s22; avg_s21]; (shape 2x10)

think – what shape would we get if we ran [avg_s22, avg_s21] ?

Creating matrices

we want to compare the grades from the class we're teaching and what we did in the past. so with means over all students in each, we can make a new matrix which has rows for semesters and columns still assignments

avg_s22 = mean(grades, 1);

avg_s21 = mean(oldgrades,1);

both_sem_grades = [avg_s22; avg_s21];

think – what shape would we get if we ran [avg_s22, avg_s21] ?

this is "implicitly" calling the **cat** command:

> cat(dim, data1, data2) will concatenate ("put together") data across the dimension you pass it, if data1 and data2 match. so
>
> **both_sem_grades = cat(1, avg_s22, avg_s21);** also works

Special matrices

we can use the commands we used with arrays with matrices too! you already saw randn, and we could use randi / rand / ones / zeros / nan the same way.

```
>> randn(3,2)

ans =

   -0.6291   -1.4286
   -1.2038   -0.0209
   -0.2539   -0.5607

>> randi(100,3,2)

ans =

    96    74
    93    27
     6    43

>> rand(3,2)

ans =

    0.5479    0.9831
    0.9427    0.3015
    0.4177    0.7011
```

```
>> ones(3,2)

ans =

     1     1
     1     1
     1     1

>> zeros(3,2)

ans =

     0     0
     0     0
     0     0

>> nan(3,2)

ans =

   NaN   NaN
   NaN   NaN
   NaN   NaN
```

Special matrices

we can use the commands we used with arrays with matrices too! you already saw randn, and we could use randi / rand / ones / zeros / nan the same way. there are a few other special commands to learn:

- **eye**: matrix with 1s along the diagonal
- **magic**: nxn matrix with values 1-n^2
- **diag**: can be used to make or get diagonals of a matrix
- **triu** and **tril**: get the upper and lower triangle of the matrix (including diag)

play around with these in your editor on this tutorial! you **will** use them in the assignments, often **together**

```
>> eye(3)

ans =

     1     0     0
     0     1     0
     0     0     1

>> diag([55 76 1:2])

ans =

    55     0     0     0
     0    76     0     0
     0     0     1     0
     0     0     0     2
```

```
>> m = magic(5)

m =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> diag(m)

ans =

    17
     5
    13
    21
     9
```

```
>> triu(m)

ans =

    17    24     1     8    15
     0     5     7    14    16
     0     0    13    20    22
     0     0     0    21     3
     0     0     0     0     9
```

```
>> tril(m)

ans =

    17     0     0     0     0
    23     5     0     0     0
     4     6    13     0     0
    10    12    19    21     0
    11    18    25     2     9
```

Special matrices

some of these commands also take an “offset” parameter. so if we want the off diagonal one up+right of the main diag, we can call with a positive 1

notice! these are different shapes.

for this 5x5 matrix, what will the size be if we call **diag(m,3)**?

also works for triu/tril

```
m =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> diag(m)

ans =

    17
     5
    13
    21
     9

>> diag(m,1)

ans =

    24
     7
    20
     3
```

```
>> tril(m,1)

ans =

    17    24     0     0     0
    23     5     7     0     0
     4     6    13    20     0
    10    12    19    21     3
    11    18    25     2     9

>> triu(m,1)

ans =

     0    24     1     8    15
     0     0     7    14    16
     0     0     0    20    22
     0     0     0     0     3
     0     0     0     0     0
```

big note – these will work with non-square matrices!

```
>> not_square = m(:,1:3)

not_square =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25

>> diag(not_square)

ans =

    17
     5
    13
```

```
>> tril(not_square)

ans =

    17     0     0
    23     5     0
     4     6    13
    10    12    19
    11    18    25
```

```
>> eye(3,7)

ans =

     1     0     0     0     0     0     0
     0     1     0     0     0     0     0
     0     0     1     0     0     0     0
```

Reshaping arrays

once we have data, we may want to change its shape

- **repmat** “repeat matrix” is for copying things in one direction or the other, *beware* passing it a single index
- **reshape** will change the data you give it to the size you request, if it is possible

```
>> v = [4 -2]

v =

     4    -2

>> repmat(v,1,2)

ans =

     4    -2     4    -2

>> repmat(v,2,1)

ans =

     4    -2
     4    -2

>> repmat(v,2)

ans =

     4    -2     4    -2
     4    -2     4    -2
```

```
>> reshape([4 -5 6 8], 2, 2)

ans =

     4     6
    -5     8

>> reshape([4 -5 6 8], 2, 10)
Error using reshape
Number of elements must not change.
```

Algebra

we can combine some of those matrix creations with what we learned about vector math. remember, **eye** creates a matrix with 1s on the diagonal and zeros everywhere else. what would the following create?

v1 = eye(3,3);

v1+v1 = ?

v1*6 = ?

3+v1 = ?

(3+v1) .* v1 = ?

Algebra

we can combine some of those matrix creations with what we learned about vector math. remember, **eye** creates a matrix with 1s on the diagonal and zeros everywhere else. what would the following create?

v1 = eye(3,3);

v1+v1 = ?

v1*6 = ?

3+v1 = ?

(3+v1) .* v1 = ?

```
>> v1 = eye(3,3)

v1 =

     1     0     0
     0     1     0
     0     0     1

>> v1+v1

ans =

     2     0     0
     0     2     0
     0     0     2

>> v1*6

ans =

     6     0     0
     0     6     0
     0     0     6
```

```
>> 3+v1

ans =

     4     3     3
     3     4     3
     3     3     4

>> (3+v1) .* v1

ans =

     4     0     0
     0     4     0
     0     0     4
```

Some other algebra…

trace will return the sum of the values on the diagonal

(so **trace(m) == sum(diag(m))**)

min and max work a bit weirdly for matrices. if you call min(m) you will get the minimum in each column. (so what would this give us for our grades example?)

if you specify 2 values in the output, you can also get the *index* of the value (for example, I don't care what the lowest value is, but I want to know which student got it)

```
>> help min
 min    Minimum elements of an array.
    M = min(X) is the smallest element in the vector X. If X is a matrix,
    M is a row vector containing the minimum element from each column. For
    N-D arrays, min(X) operates along the first non-singleton dimension.

    When X is complex, the minimum is computed using the magnitude
    min(ABS(X)). In the case of equal magnitude elements the phase angle
    min(ANGLE(X)) is used.

    [M,I] = min(X) also returns the indices corresponding to the minimum
    values. The values in I index into the dimension of X that is being
    operated on. If X contains more than one element with the minimum
    value, then the index of the first one is returned.
```

```
m =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> min(m)

ans =

     4     5     1     2     3
```

Some other algebra...

if you want the min across a specific dimension, you call [] as the second input and then either ‘all’ or DIM as the third (so to get the min across rows, you would pass in [], 2).

if you pass a single value without the [], matlab thinks you want to compare at each place

C = **min**(X,Y) returns an array with the smallest elements taken from X or Y. X and Y must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

M = **min**(X,[],'all') returns the smallest element of X.

[M,I] = **min**(X,[],'all') also returns the linear index into X that corresponds to the minimum value over all elements in X.

M = **min**(X,[],DIM) or [M,I] = **min**(X,[],DIM) operates along the dimension DIM.

```
>> min(m,10)

ans =

    10    10     1     8    10
    10     5     7    10    10
     4     6    10    10    10
    10    10    10    10     3
    10    10    10     2     9
```

```
m =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> min(m)

ans =

     4     5     1     2     3

>> min(m,[],1)

ans =

     4     5     1     2     3

>> min(m,[],2)

ans =

     1
     5
     4
     3
     2
```

Logic and matrices!

let's return to our class grades example. remember, I have a 5x9 matrix **grades** to work with, for 5 students and 9 assignments *(three homework assignments and a test for each of 2 semesters, plus a final*). I want to find out which of my students got As and which got Bs – I will use just their two tests and final to make this decision, and I want to find who has above 90 and who has below 90 average for those assignments. how can I do this?

try thinking through each step! 1) select assignments, 2) combine assignments, 3) compare to grade cutoff

Logic and matrices!

let's return to our class grades example. remember, I have a 5x9 matrix **grades** to work with, for 5 students and 9 assignments *(three homework assignments and a test for each of 2 semesters, plus a final*). I want to find out which of my students got As and which got Bs – I will use just their two tests and final to make this decision, and I want to find who has above 90 and who has below 90 average for those assignments. how can I do this?

try thinking through each step! 1) select assignments, 2) combine assignments, 3) compare to grade cutoff

a_students = mean(grades(:,[4 8 9]), 2) > 90

Practice (if time)

1. create a 5x5 matrix z which has 8 on the diagonal and -2 everywhere else
2. set n to the sum of the elements in the even numbered rows
3. set m to a vector of the max value in each row of z
4. make a new matrix z2 which has m*3 as its first column, followed by z