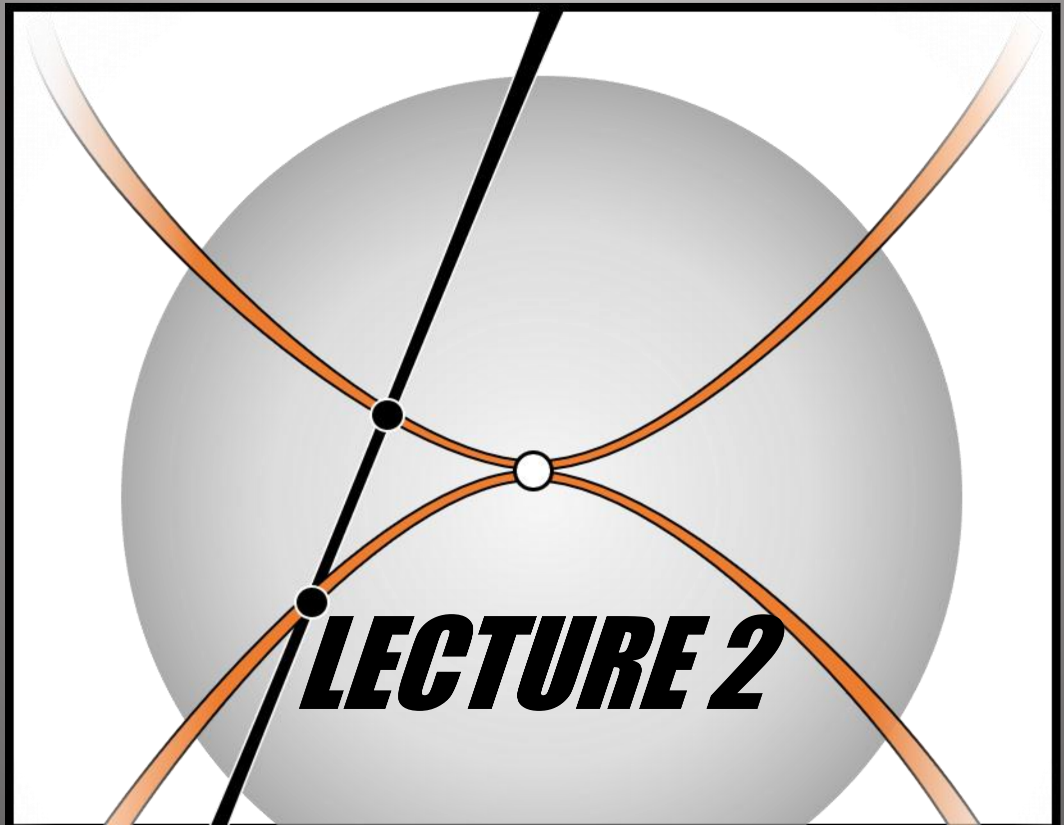
COMPUTATIONAL MATHEMATICS

Vidit Nanda
nanda@maths.ox.ac.uk



OUTLINE

- Arrays
- Logic
- Programs

ARRAYS

Arrays are grids of numbers/variables/(insert favourite object here)

One-dimensional arrays are *vectors* (grid = 1 x n)

Two-dimensional arrays are *matrices* (grid = m x n)

The hierarchy keeps going, but let's stop here for now.

Arrays can be created by using [blah blah moreblah], with semicolons to indicate "go to the next row"

```
>> v = [1 2 -1 9]
v =
    1 2 -1 9
```

```
>> A = [1 2; -1 9]
A =
       1   2
      -1   9
```

If you try [1 2 -1; 9] then Matlab will yell at you (loudly, and for good reasons).

CONCATENATION AND RANGES

Combining arrays horizontally and vertically is easy:

```
>> [v v]
ans =
    1 2 -1 9 1 2 -1 9
```

```
>> [v ; v]
ans =
    1 2 -1 9
    1 2 -1 9
```

For assigning equally-spaced numbers, use *start* : *step* : *stop*, eg

```
>> r = 1 : 0.1 : 1.55
r =
    1.000 1.100 1.200 1.300 1.400 1.500
```

Just using *start* : *stop* assumes *step* = 1;

Negative steps are allowed

With great power comes great responsibility: don't do 1 : 0.1: -2

ARRAY MANIPULATION 1

Here are some common things to do with arrays:

```
>> transpose(A) or >> A.' gives the transpose
>> conj(A) is the complex conjugate
>> A' is the Hermitian conjugate = transpose(conj(A))
>> inv(A) is the inverse matrix (if one exists!)
>> A\b solves Ax = b, and >> b/A solves xA = b
>> det(A) is the determinant
```

The standard +, - and * operations work directly for matrices (**provided the sizes match** as expected)

But *behold this affront to common decency*: matrix + scalar acts component-wise

```
>> A + 1
ans =
      2   3
      0  10
```

```
>> A - 1
ans =
      0   1
     -2   8
```

ARRAY MANIPULATION 2

Generally, to perform a basic operation elementwise, we have to *preface it with a leading dot* (.) like this:

```
>> A.^2
ans =
      1   4
      1  81
```

```
>> A.^0.5
ans =
   1.0000 + 0.0000i   1.4142 + 0.0000i
   0.0000 + 1.0000i   3.0000 + 0.0000i
```

But common functions (sin, tan, exp, log,…) already work elementwise:

```
>> cos(A.^2)
ans =
    0.5403   -0.6536
    0.5403    0.7767
```

```
>> log(A)
ans =
   0.0000 + 0.0000i   0.6931 + 0.0000i
   0.0000 + 3.1416i   2.1972 + 0.0000i
```

The most commonly used built-in functions for manipulating arrays are:

sort(A,d)	sort A along dimension d
repmat(A,m,n)	concat. A with itself, m horizontal & n vertical copies
reshape(A,m,n)	reshape A into an m x n matrix

ARRAY ACCESS

Vector access requires **numbers/ranges within parentheses**:

```
>> v(3)
ans =
      -1
>> v(end)
ans =
      9
```

```
>> v(2:end);
ans =
      2 -1 9
>> v(end-1:end)
ans =
      -1 9
```

For matrices, it's **comma-separated pairs** of numbers/ranges:

```
>> A(1,2)
ans =
      2
>> A(1,:)
ans =
      1    2
```

```
>> A(:,1)
ans =
      1
     -1
```

```
>> A(3)
ans =
     2
```

????

BONUS! USEFUL SHORTCUTS

eye(n)	n x n eye-dentity matrix
zeros(m,n)	this should be obvious
ones(m,n)	this also
rand(m,n)	m x n matrix with uniformly distributed entries in [0,1]
rand(m)	same as above, but n = m
randn(m,n)	normally distributed, i.e., N(0,1) entries
randn(m)	same as above, but with n = m
diag(v)	diagonal matrix with diagonal vector v

sum(A,d)	sum-vector along dimension d
prod(A,d)	product-vector along dimension d
size(A)	size vector of A, i.e., [m n] for m x n matrix
max(A)	same as above, but n = m
length(v)	length of vector v
max(A)	vector of maximum entries along columns of A

LOGIC

In Matlab, as in many other languages, there are **logical variables** which evaluate to 0 (false) or 1 (true)

```
>> x = 3
>> x < 7
ans =
      logical 1
```

```
>> v = [3 2 -1];
>> length(v) = 4
ans =
      logical 0
```

And you get **logical arrays** by evaluating conditionals component-wise, eg:

```
>> [1 3; 2 4] > [2 3; 0 0]
ans =
  2×2 logical array
   0   0
   1   1
```

```
>> [1 3; 2 4] <= 2
ans =
  2×2 logical array
   1   0
   1   0
```

Other important comparisons: = = checks **equality**, ~ = checks **not-equality**

LOGICAL INDEXING

We can use logical arrays to **find interesting stuff** (that satisfies chosen constraints) within other arrays. Eg, *to find all the positive even numbers*:

```
>> v = [1 3 -4 2 7 12 9 -18 6 19]
>> v((mod(v,2) == 0) & v > 0)
ans =
     2    12     6
```

A lot has happened in this one line!

First, a logical array is made for all entries in v whose remainder mod 2 is 0:

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

Another one is made for all the positive entries

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

The "bit-wise and" operation, i.e., multiplication, is performed componentwise

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

And *finally*, the entries in v corresponding to the 1 positions are selected

Less slick, but easier for humans to read: **find**(v) is the same as (v > 0)

PROGRAMS

For commonly-needed tasks that don't already have a built-in function, you can **write your own functions** and call them from the >> … prompt

Functions are written into **.m-files**, each one looks like this:

```
% comment explaining what this function does
function [out1, out2,…] = funcName(in1, in2,…)
      statement 1
      statement 2
      out1 = …
      out2 = …
end
```

The first non-commented line is the *signature* of the function, which defines all the input and output variables (along with the name --- the file is **funcName.m**)

The stuff in the body of the function can get complicated and difficult to keep track of: as a favour to your future self (and others), **please comment generously**!

PROGRAMS: CONTROL FLOW

Three basic keywords will help organise the complicated interior of your programs: **if, for** and **while.** Here's **if:**

Optional! (the `else` part)

```
if det(A) ~= 0
      B = inv(A);
else
      display('curses, foiled again!');
end
```

For more elaborate decision-making, you can string these into longer conditionals:

Can have many! (the `elseif` part)

```
if det(A) ~= 0
      B = inv(A);
elseif A > 0
      display('think positive!');
else
      display('definitely foiled.');
end
```

PROGRAMS: CONTROL FLOW

The **for** loop is for when you want to perform a task a known number of times:

```
for i=1:100
     display('my code has no comments. shame!');
end
```

And **while** is used when you don't know how many times to loop:

```
n = 20;  % total number of prime numbers needed
i = 2;   % starting point of search
primes = zeros(n,1); % this stores the answer
while n > 0
     if isprime(i) % check prime-ness
          n = n-1; % need one fewer prime now
          primes(end-n) = i; % fill from the beginning
     end
     i = i+1; % now to check next number
end
```