Computational Mathematics

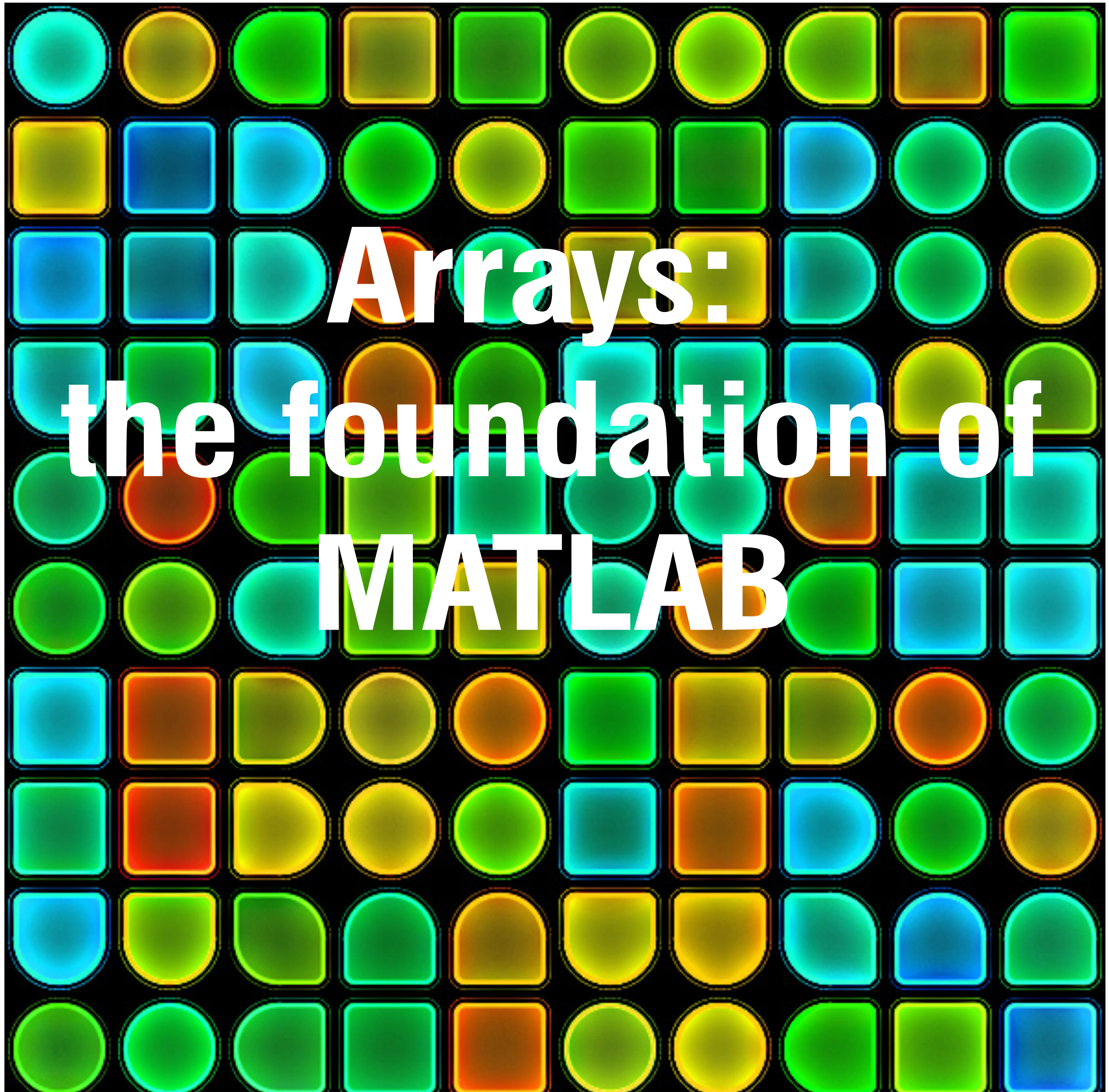
Michaelmas Term Lecture 2

Andrew Thompson

Outline for today:

- Arrays
- Logic
- Programming
- Functions

Arrays: the foundation of MATLAB



Arrays

MATLAB is very good at dealing with arrays

A vector is a 1d array; a matrix a 2d array

Arrays with more dimensions are allowed, but uncommon

Construct a row vector like so:

```
>> a = [1 2 3 4]
a =
     1     2     3     4
```

Enter a 2-by-2 matrix like this

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
```

N.B. MATLAB is case sensitive, so a and A are different variables.

Concatenation

Note that the semicolon was used to separate two rows of the matrix

The semicolon works as a concatenation operator

It can be used to concatenate two arrays in the up-down direction:

```
>> a
a =
     1     2     3     4
>> [a;a]
ans =
     1     2     3     4
     1     2     3     4
```

The space concatenates in the left-right direction:

```
>> [A A]
ans =
     1     2     1     2
     3     4     3     4
```

Ranges

Often we require a vector of equally spaced numbers

MATLAB has *ranges* to deal with this

Declare a range with `startvalue:stopvalue`

```
>> r = 1:10
r =
     1     2     3     4     5     6     7     8     9    10
```

Ranges need not have integral spacing: use `startvalue:step:stopvalue`

```
>> r = 1:0.2:2
r =
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
r = 2:-0.2:1
    2.0000    1.8000    1.6000    1.4000    1.2000    1.0000
```

Array manipulation

Matrix transpose:	`transpose(a)` or `a.'`
Complex conjugate:	`conj(a)`
Hermitian transpose:	`a'`
Inverse:	`inv(a)`
Left matrix division (solve **Ax=b**)	`A\b`
Right matrix division (solve **xA=b**)	`b/A`
Determinant:	`det(a)`

Left and right matrix division are much more efficient than using `inv`

Array arithmetic

For matrices `*` is interpreted as matrix multiplication

`+` and `-` work for matrices

Addition of a matrix and a scalar is interpreted sensibly:

```
>> [1 2 3] + 1
ans =
     2     3     4
```

Elementwise operations

There are occasions when we wish operations to act on each element of a matrix, rather than the whole matrix.

Example: computing the square of every element of a matrix `squareMat`:

`squareMat^2` is not what is required.

To make an operator act *elementwise,* prefix it with a dot:

```
squareMat.^2
```

Another example: consider vectors `x` and `y`:

```
x./y + y.^2 -2*y.*x
```

Most of the mathematical functions covered work with arrays elementwise:

```
>> sin([0 pi/4 pi/3 pi/2 pi])
ans =
     0.0000    0.7071    0.8660    1.0000    0.0000
```

`exp` works elementwise: use `expm` for matrix exponentials

Array construction functions

MATLAB has many functions to construct common matrices:

`eye(n)`	n-by-n identity matrix
`zeros(m,n)`	m-by-n zero matrix
`ones(m,n)`	m-by-n matrix of ones
`rand(m,n)`	uniformly distributed m-by-n matrix
`randn(m,n)`	$N(0,1)$ distributed m-by-n matrix
`diag(x)`	diagonal matrix formed using vector x

Array access

Vectors are accessed using a single subscript between brackets:

```
>> v =[1 3 5];
>> v(3)
ans =
     5
>>v = v.';
>> v(2)
ans =
     3
```

Matrix elements are accesed using the row and column number:

```
>> A = [1 2;3 4];
>> A(2,2)
ans =
4
```

Array access continued

The word end can be used to refer to the last element along a dimension:

```
>> x = 1:100;
>> x(end)
ans =
     100
```

Ranges can be used to access arrays:

```
>> x(1:5)
ans =
     1     2     3     4     5
```

A more complicated example:

```
>> A = [1 2 3;4 5 6;7 8 9];
>> A(2,1:end)
ans =
     4     5     6
```

Functions for array manipulation

`repmat(A,m,n)`	concatenate A m times vertically, n times horizontally
`reshape(A,m,n)`	reshape the elements of A into an m-by-n matrix
`sort(A,dim)`	sort A along the dimension `dim`
`flipud(A)`	flip A in the up-down direction
`fliplr(A)`	flip A left-to-right
`circshift(A,n)`	circularly shift elements of A down by an amount n

Functions that interrogate arrays

`sum(A,dim)`	sum elements of A along dimension dim
`prod(A,dim)`	form product of elements of A along dimension dim
`size`	return vector of dimensions of A
`length`	return length of vector
`numel`	return number of elements of an array
`nnz`	return number of elements not equal to
max	return maximum of each column

Logic

Logical expressions

We have met some variable classes already: string, integer, double precision

MATLAB has another for handling logic: the *logical* class

A logical variable can have the value `true` or `false`

```
>> x = true
x =
    1
>> class(x)
ans =
     logical
```

True and false are also represented by `1` and `0`:

```
>> x = logical(0); % sets x to false
```

Logical expressions: comparison

We can make *logical comparisons* in MATLAB

```
>> 2 > 1
ans =
     1
>> 1 == 0
ans =
     0
```

==	is equal to
~=	is not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Array logic

All logical expressions covered so far work with arrays *elementwise*

The result is an *array of logical values* (0s or 1s); a *logical array*

Here we see arrays being compared:

```
>> A = [1 2;3 4]; B = [1 2;-3 4];
>> A == B
ans =
     1     1
     0     1
```

We may perform Boolean operations with logical arrays as well:

```
>> a = logical([1 0 0]); b = logical([0 1 0])
>> a | b
ans =
     1     1     0
```

Logical indexing: powerful expressions

We may use a logical array to index another array

Why is this useful?

Suppose we wish to find all numbers in a matrix fulfilling some criteria

e.g. **all the positive entries**

We write an expression whose result is a logical array:

```
>> z = [1 2 -1 0 -4 20 -2];
>> z > 0
ans =
     1    1    0    0    0    1    0
```

Use this array **to index the original array**:

```
>> index = z > 0;
>> z(index)
ans =
     1    2    20
```

It is usually much neater to write a single expression:

```
>> z(z>0)
ans =
     1     2    20
```

A more complicated example: return all the elements that are on the diagonal:

```
>> a = 1:16;
>> A = reshape(a,4,4); % create a 4-by-4 matrix
>> index = logical(eye(4))
>> A(index)
ans =
     1     6    11    16
```

What will the following return from a matrix X?

```
X((mod(X,2)==0) & (X > 0))
```

The `find` function

The find function returns **indices of the nonzero elements of an array**

This is useful to find the *indices* of elements that fulfil certain criteria

Using `find`

```
>> a = [1 0 5 0 -1]
>> find(a)
ans =
     1    3    5
```

Combine find with a logical expression:

```
>> find(a < 0)
ans =
     5
```

The M-file

Getting started

We can write programs or *scripts* for MATLAB

At their simplest these are a list of statements one after another

Written in an M-file, using the `.m` extension

No special structure: simplest program is just a list of statements

A simple code:

```
% simple.m

A = rand(2);
display(eig(A));
```

Loops

- “For” loop
  - Typically, one knows how many terms
  - Example: sum of n^2 for n=1..10
- “While” loop
  - Use when it's not obvious how many terms are needed
  - Example: find first 10 prime numbers
- Flow control: “if ... then ... else ...”

Program flow: `if` statements

We can control whether certain parts of a program are executed

We can make execution conditional using an `if` statement

An example: compute a matrix inverse only if matrix is nonsingular:

```
if (abs(det(A)) > eps)
    display(inv(A));
end
```

We can allow the program to follow one of two paths using the `else` keyword:

Example: display a warning if the matrix is singular:

```
if (abs(det(A)) > eps)
    display(inv(A));
else
   display('matrix is singular to working precision')
end
```

The **elseif** keyword

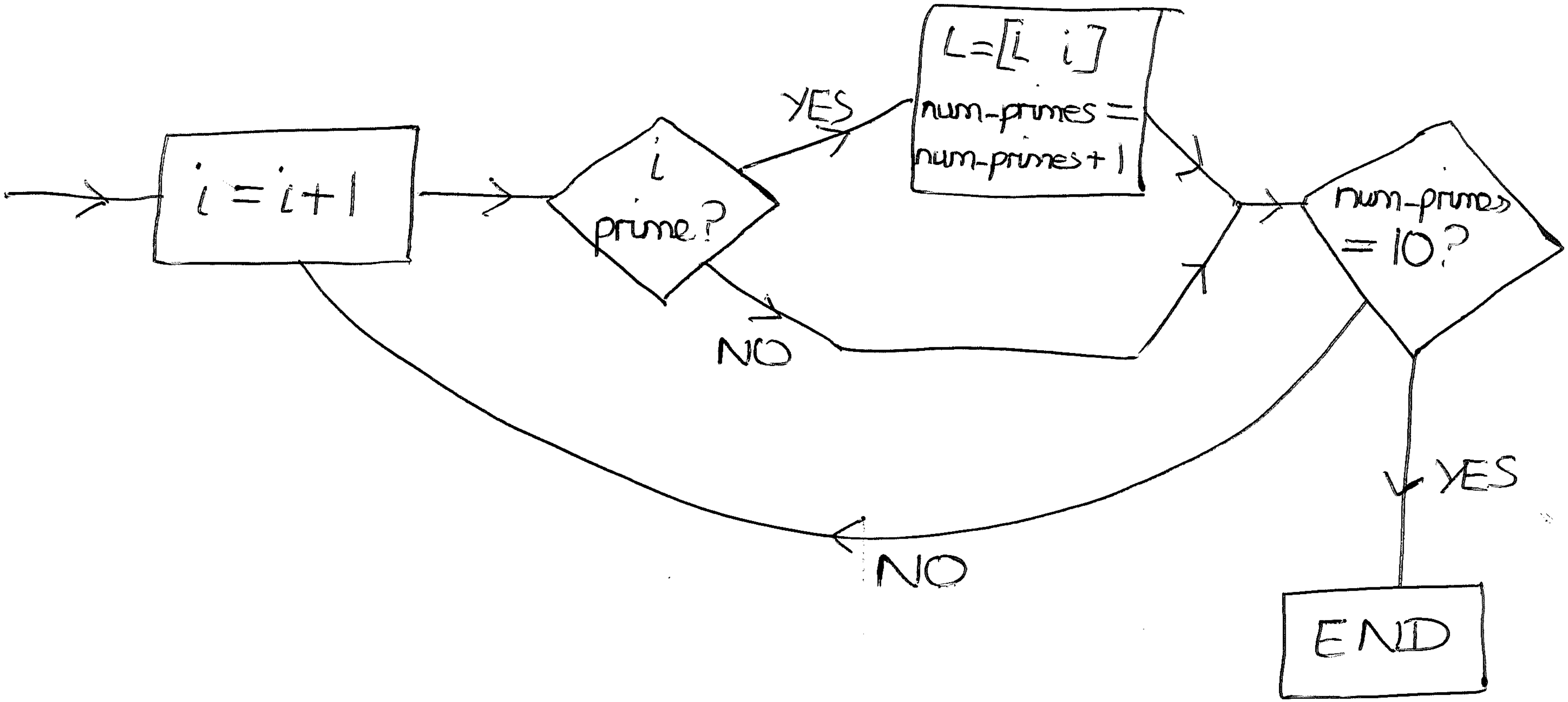
The **elseif** keyword allows the program to follow one of several branches

Example: display a message about the size of a 2d array

```
% part of a program
x = min(size(A));
if (x==0)
    display('A is empty');
elseif(x==1)
     display('A is a vector');
else
    display('A is a matrix');
end
```

We used **else** here to catch all the other possible cases

N.B. spelling of elseif vs elsif as in some languages (Ruby, Perl)



i = i+1
i
prime?
YES
NO
L=[L i]
num_primes =
num_primes+1
num_primes
=10?
YES
NO
END

Programming

- Why? Sometimes problems are too “hard” for a sequential approach:
  - Algorithms
  - Repetition, encapsulation, “code reuse” (solve a low-level problem once, make sure its correct, use it repeatedly without worrying about the details)
  - Example: convert your problem to linear algebra: Ax=b, then call solver.
  - Build bigger ideas on top of smaller ones, e.g. “isprime()”
- Its fun! Like Lego (well, some people think so).
- Many careers involve solving problems on computers... and typically, this means programming.

Writing Functions

- Take some input, give back some output
- You already know how to write inline 'anonymous' functions, e.g. f = @(x) x^2 - 4*x
- Matlab already has many built-in functions
  - often implementations of mathematical functions y = f(x)
  - can be instructive to ask how these work (sometimes you can read the source code, but unfortunately not always)
- [Also symbolic functions: algebraic objects]

Function files

Writing your own functions

We may add to the many MATLAB built-in functions

Simply write a function and save in an M-file, e.g MyFunction

Call the function in the normal way

```
>> MyFunction
```

MATLAB searches for the function in the current directory and executes it

Functions are also written in a `.m` file

Function structure

Functions all have the same structure

You can even look at the code for the built-in functions

A skeleton function:

```
function [out1,out2,...] = functionName(arg1,arg2,...)
    statements
     ⋮
    out1 =
    out2 =

end
```

First line is the function signature

Result/output variables are defined within the function

Function ends with an **end** (actually optional, but a good idea)

Simple functions

Example: some simple functions

```
function [] = proclaim()
   display('MATLAB is awesome');
end
```

```
function [xout] = jukowski(xin)
   xout = xin + 1./xin;
end
```

Call one function from another:

```
function [xout] = jukowski(xin)
   xout = xin + 1./xin;
   proclaim();
end
```