Introduction to MATLAB

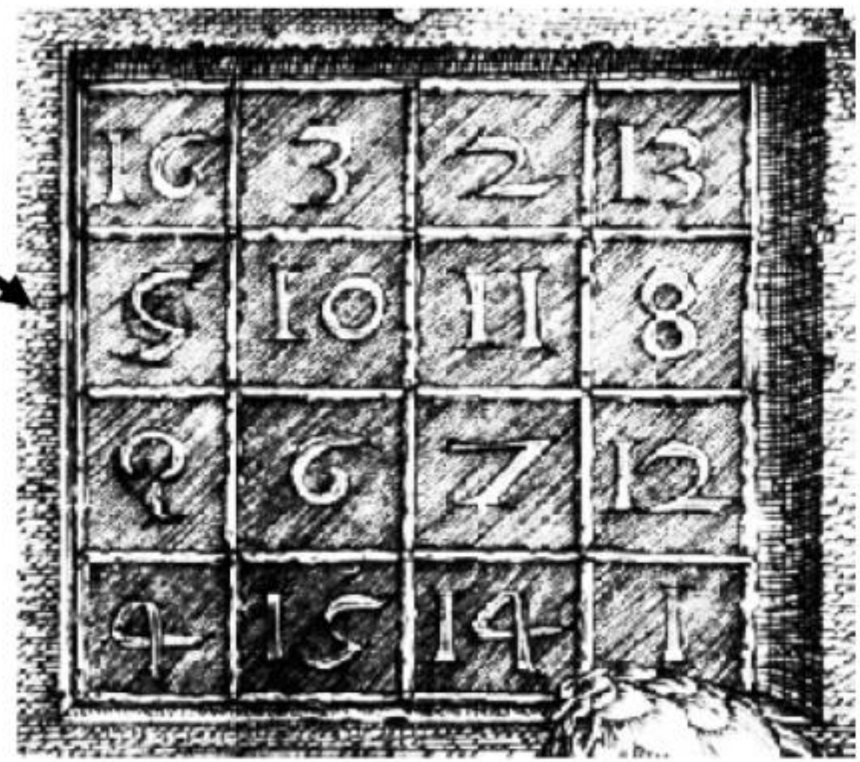
Bio-Medical Computing Lab, BMCL

Introduction to MATLAB

- **MATLAB** is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment. Typical uses include:
  - Math and computation
  - Algorithm development
  - Modeling, simulation, and prototyping
  - Data analysis, exploration, and visualization
  - Scientific and engineering graphics

- **MATLAB** is an ***interactive*** system whose basic data element is an ***array*** that does not require dimensioning. This allows you to solve many technical computing problems, especially those with ***matrix*** and ***vector*** formulations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C or Fortran.

Entering Matrices (1) – Magic square



- Engraving by Albrecht Dürer, German artist and mathematician in 1514.

Entering Matrices (2) – Method 1:Direct entry

- **4 ways of entering matrices in MATLAB:**
  - Enter an explicit list of elements
  - Load matrices from external data files
  - Generate matrices using built-in functions
  - Create matrices with your own functions in M-files

- **Rules of entering matrices:**
  - Separate the elements of a row with ***blanks*** or commas
  - Use a ***semicolon*** ";" to indicate the end of each row
  - Surround the entire list of elements with ***square brackets***, [ ]

- To enter Dürer's matrix, simply type:

  » A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]

- MATLAB displays the matrix you just entered

```
EDU>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]

A =

    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

Entering Matrices (3) – as lists

- Why is this a magic square? Try this in Matlab

```
» sum(A)
ans =
    34    34    34    34
» A'
ans =
    16     5     9     4
     3    10     6    15
     2    11     7    14
    13     8    12     1
» sum(A')'
ans =      34
           34
           34
           34
```

Compute the sum of each column in A

Result in row vector variable ans

Transpose matrix A

Compute the sum of each row in A

Result in column vector variable ans

Entering Matrices (4) – subscripts

- A(i,j) refers to element in row i and column j of A

row

col

```
» A(4,2)
ans = 15
» A(1,4) + A(2,4) + A(3,4) + A(4,4)
ans = 34
» X = A;
» X(4,5) = 17
X =
     16     3     2    13     0
      5    10    11     8     0
      9     6     7    12     0
      4    15    14     1    17
```

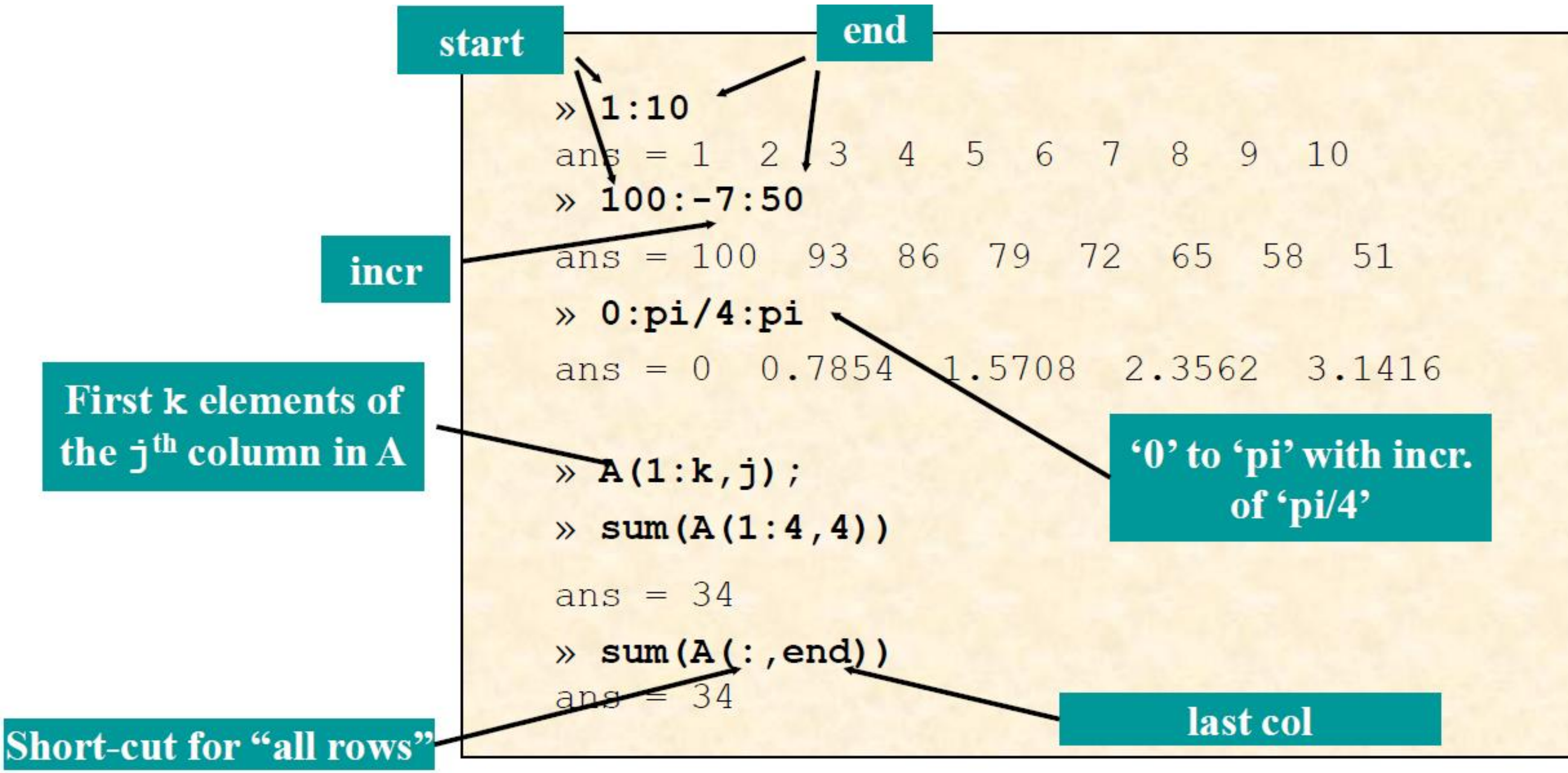
Slow way of finding sum of column 4

Make another copy of A in X
';' suppress output

Add one element in column 5, auto increase size of matrix

Entering Matrices (5) – colon : operator

- ':' used to specify range of numbers



Expressions & built-in functions

Elementary functions

```
» rho = (1+sqrt(5))/2
rho =  1.6180
```

Complex number

```
» a = abs(3+4i)
a =  5
```

Special functions

```
» z = sqrt(besselk(4/3,rho-i))
z = 0.3730+ 0.3214i
```

Built-in constants (function)

```
» huge = exp(log(realmax))
huge =  1.7977e+308

» toobig = pi*huge
toobig = Inf
```

- `pi` — 3.14159265
- `I or j` — Imaginary unit, -1
- `eps` — FP relative precision, 2^{-52}
- `realmin` — Smallest FP number, 2^{-1022}
- `realmax` — Largest FP number, $(2-)2^{1023}$
- `Inf` — Infinity
- `NaN` — Not-a-number

Entering Matrices (6) – Method 2: Generation

```
» Z = zeros(2,4)
Z =   0       0       0       0
      0       0       0       0

» F = 5*ones(3,3)
F =   5       5       5
      5       5       5
      5       5       5

» N = fix(10*rand(1,10))
N = 4       9       4       4

» R = randn(4,4)
R =   1.0668   0.2944   0.6918  -1.4410
      0.0593  -1.3362   0.8580   0.5711
     -0.0956   0.7143   1.2540  -0.3999
     -0.8323   1.6236  -1.5937   0.6900
```

Useful Generation Functions

- `zeros` All zeros
- `ones` All ones
- `rand` Uniformly distributed random elements between (0.0, 1.0)
- `randn` Normally distributed random elements, mean = 0.0, var = 1.0
- `fix` Convert real to integer

Entering Matrices (7) – Method 3&4: Load & m-file

magik.dat

```
16.0  3.0  2.0 13.0
 5.0 10.0 11.0  8.0
 9.0  6.0  7.0 12.0
 4.0 15.0 14.0  1.0
```

```
» load magik.dat
```

Read data from file into variable `magik`

Three dots (…) means continuation to next line

```
» magik
```

.m files can be run by just typing its name in Matlab

magik.m

```
A = [ ...
16.0  3.0  2.0 13.0
 5.0 10.0 11.0  8.0
 9.0  6.0  7.0 12.0
 4.0 15.0 14.0  1.0];
```

Entering Matrices (8) – Concatenate & delete

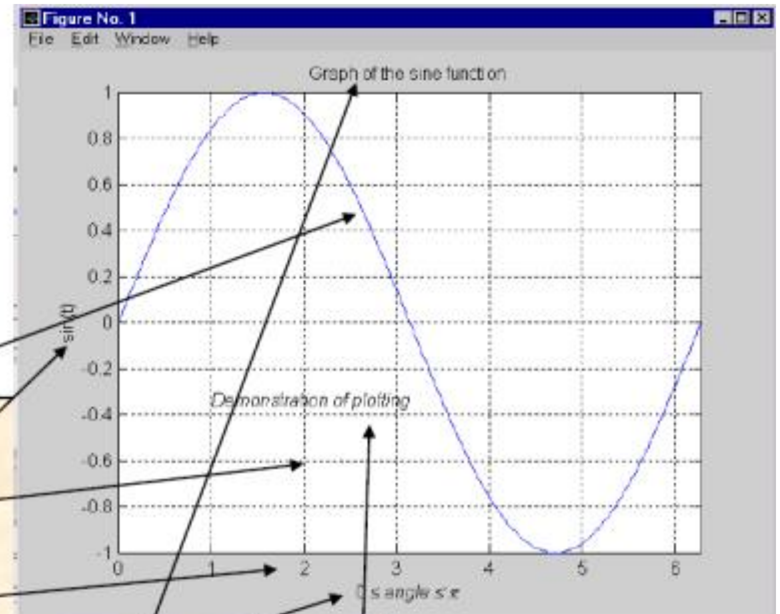
```
» B = [A  A+32; A+48  A+16]
B =
   16    3    2   13     48   35   34   45
    5   10   11    8     37   42   43   40
    9    6    7   12     41   38   39   44
    4   15   14    1     36   47   46   33

   64   51   50   61     32   19   18   29
   53   58   59   56     21   26   27   24
   57   54   55   60     25   22   23   28
   52   63   62   49     20   31   30   17
```

2nd column deleted

```
» X = A;
» X(:,2) = []
X =
    16     2    13
     5    11     8
     9     7    12
     4    14     1
```

MATALB Graphics (1) – Creating a Plot




```
» t = 0:pi/100:2*pi;
» y = sin(t);
» plot(t,y)
» grid
» axis([0 2*pi -1 1])
» xlabel('0 \leq \itangle \leq \pi')
» ylabel('sin(t)')
» title('Graph of the sine function')
» text(1,-1/3,'\it{Demonstration of plotting}')
```

MATLAB Programming (1)

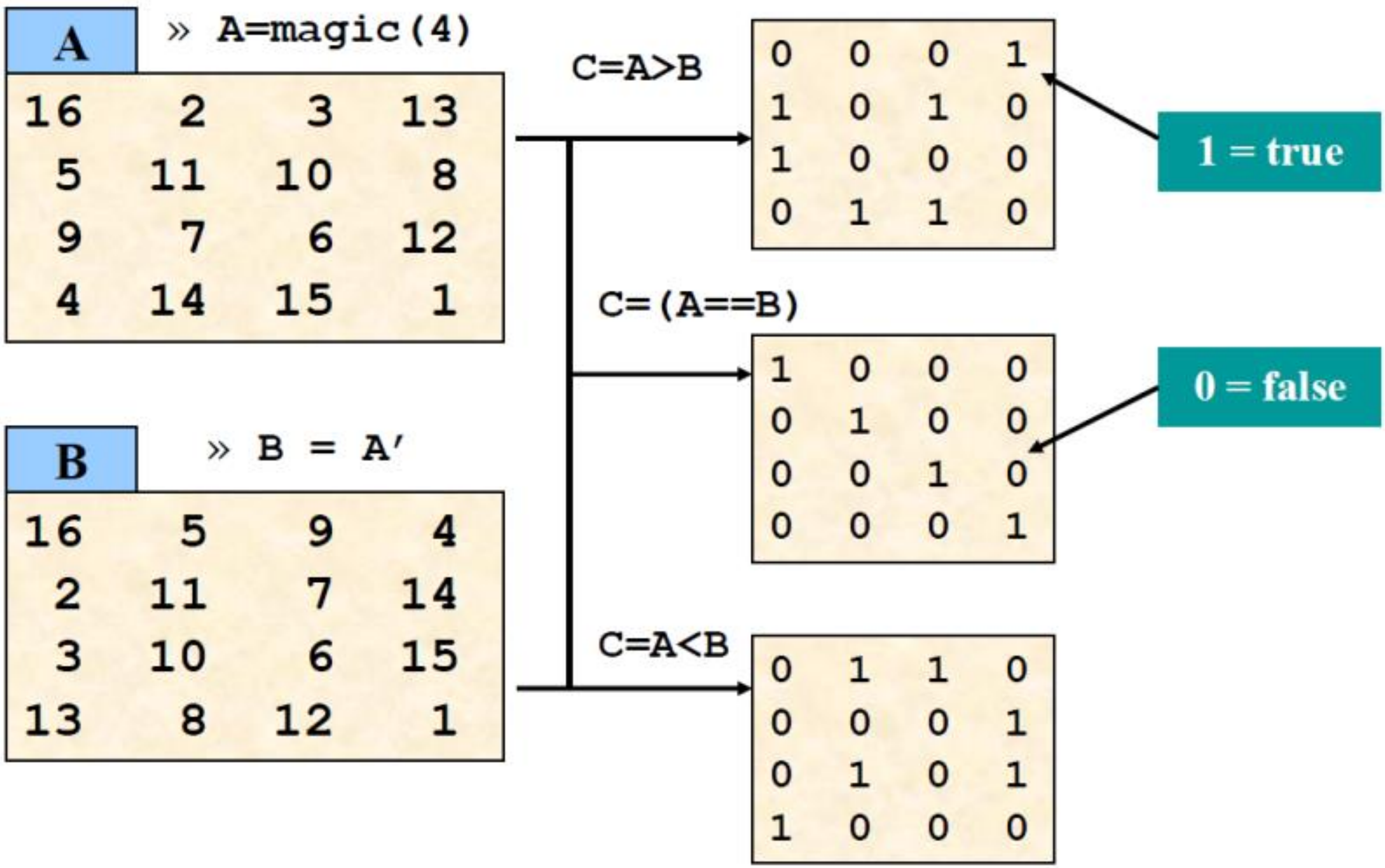
- MATLAB has five flow control constructs:
  - **if** statements
  - **switch** statements
  - **for** loops
  - **while** loops
  - **break** statements

- **if** statement

```
if A > B
   'greater'
elseif A < B
   'less'
elseif A == B
   'equal'
else
   error('Unexpected situation')
end
```

>, < and == work with *scalars*, but NOT matrices

MATLAB Programming (2) - Comparison



MATLAB Programming (3) – Built-in Logic functions for matrices

- Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with if, including
  - **isequal(A,B)** returns ‘1’ if A and B are identical, else return ‘0’
  - **isempty(A)** returns ‘1’ if A is a null matrix, else return ‘0’
  - **all(A)** returns ‘1’ if ***all*** elements A is non-zero
  - **any(A)** returns ‘1’ if ***any*** element A is non-zero

```
if isequal(A,B)
    'equal'
else
    'not equal'
end
```

MATLAB Programming (3) – Control Flow -Switch & Case

- Assume `method` exists as a string variable:

```
switch lower(method)
    case {'linear','bilinear'}
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
    case 'nearest'
        disp('Method is nearest')
    otherwise
        disp('Unknown method.')
end
```

Use otherwise to catch all other cases

MATLAB Programming (3) – Control Flow -For Loop

This makes it faster and use less memory

```
n = 4;
a = zeros(n,n)   % Preallocate matrix
for i = 1:n
    for j = 1:n
        H(i,j) = 1/(i+j);
    end
end
```

"Life is too short to spend writing for-loops"

- Create a table of logarithms:

```
x = 0;
for k = 1:1001
   y(k) = log10(x);
   x = x + .01;
end
```

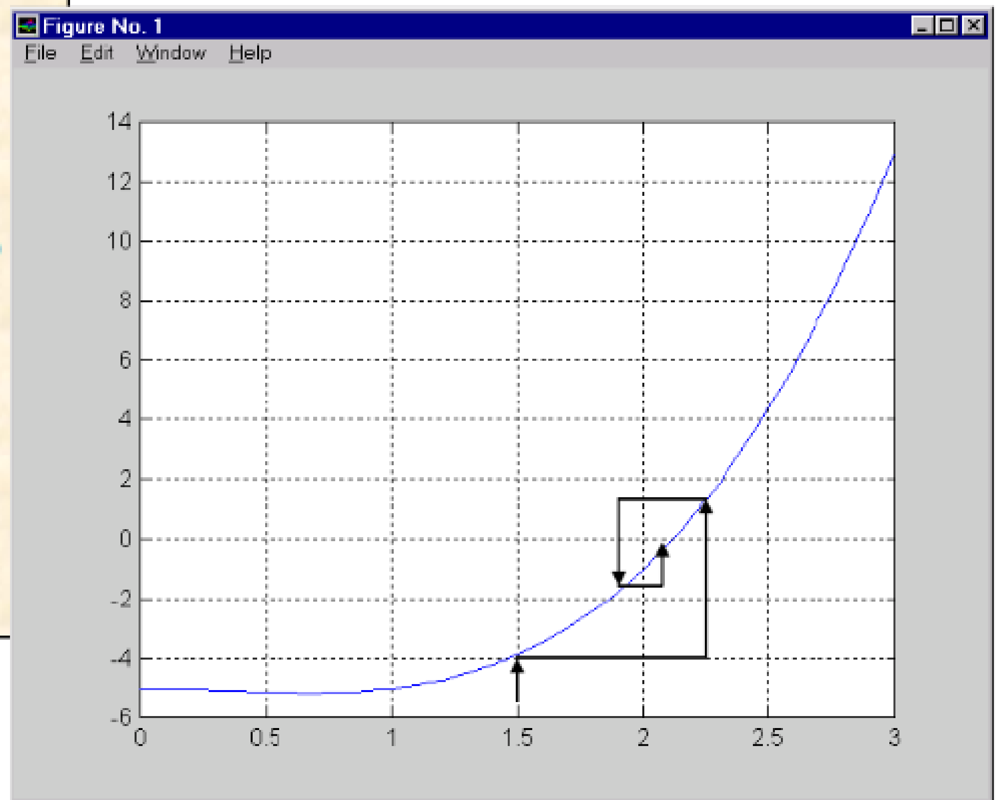
- A vectorized version of the same code is

```
x = 0:.01:10;
y = log10(x);
```

MATLAB Programming (3) – Control Flow -While Loop

```
eps = 0.0001
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
x
```

Find root of the polynomial $x^3 - 2x - 5$ ..



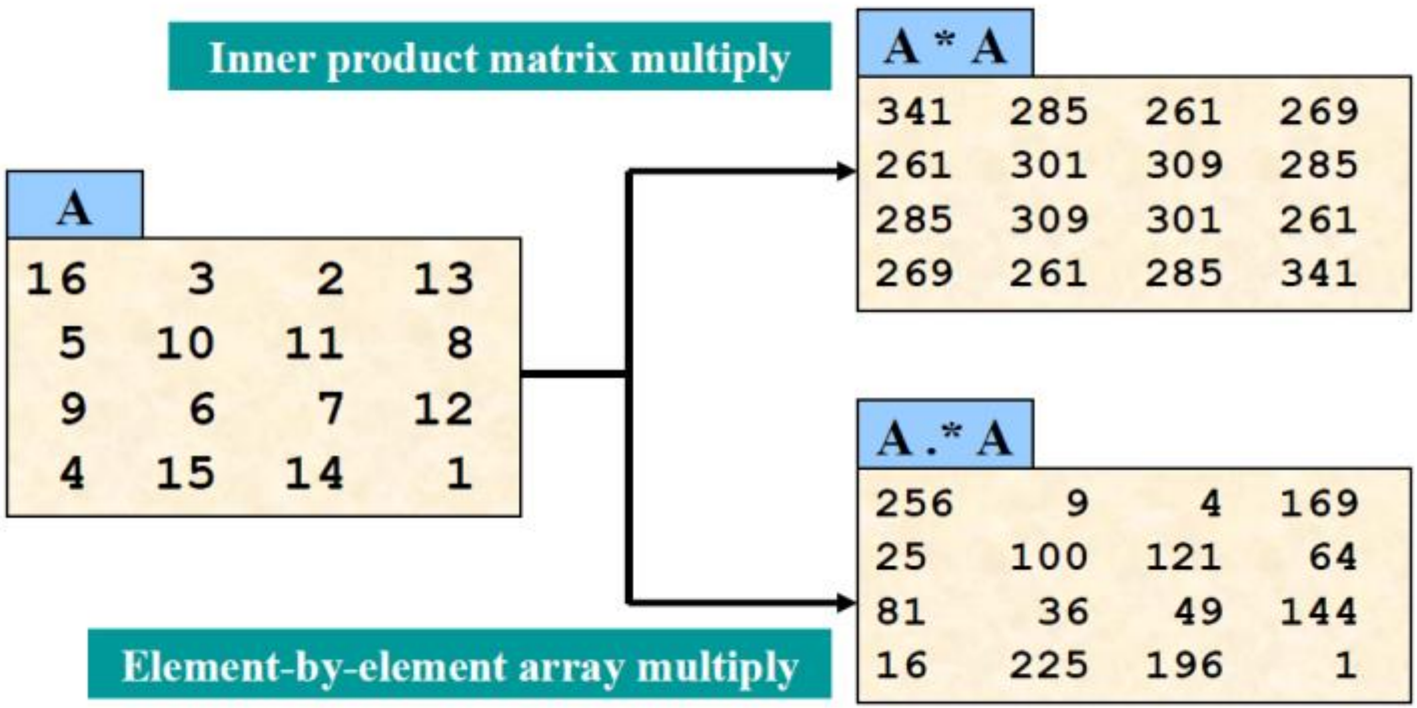
… using iterative bisection method

MATLAB Programming (3) – Control Flow -Break

- The **break** statement lets you exit early from a for or while loop.
- In nested loops, break exits from the innermost loop only.
- Is this version of the bisection program better?

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if fx == 0
      break
   elseif sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
x
```

Matrix versus Array Operations



Matrix Operations

+	Addition or unary plus. A+B adds A and B. A and B must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.
-	Subtraction or unary minus. A-B subtracts B from A. A and B must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.
*	Matrix multiplication. C = A*B is the linear algebraic product of the matrices A and B. For nonscalar A and B, the number of columns of A must equal the number of rows of B. A scalar can multiply a matrix of any size.
/	Slash or matrix right division. B/A is roughly the same as B*inv(A). More precisely, B/A = (A'\B')'. See \.
\	Backslash or matrix left division. If A is an n-by-n matrix and B is a column vector with n components, or a matrix with several such columns, then X = A\B is the solution to the equation $AX = B$.
^	Matrix power. X^p is X to the power p, if p is a scalar. If p is an integer, the power is computed by repeated multiplication.
'	Matrix transpose. A' is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.

Array Operations

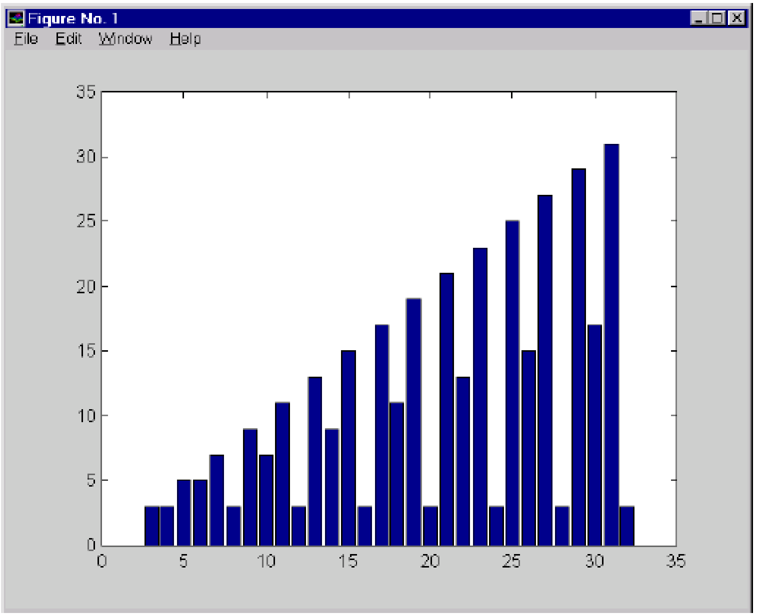
+	Element-by-element addition or unary plus.
-	Element-by-element subtraction or unary minus.
.*	Array multiplication. `A.*B` is the element-by-element product of the arrays `A` and `B`. `A` and `B` must have the same size, unless one of them is a scalar.
./	Array right division. `A./B` is the matrix with elements `A(i,j)/B(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.
.\	Array left division. `A.\B` is the matrix with elements `B(i,j)/A(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.
.^	Array power. `A.^B` is the matrix with elements `A(i,j)` to the `B(i,j)` power. `A` and `B` must have the same size, unless one of them is a scalar.
.'	Array transpose. `A.'` is the array transpose of `A`. For complex matrices, this does not involve conjugation.

M-files: Scripts and Functions

- There are two kinds of M-files:
  - Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
  - Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

Script magic_rank.m

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
   r(n) = rank(magic(n));
end
r
bar(r)
```



Functions

Return variable

Define function name and arguments

function myfunct.m

```
function r = myfunct (x)
% Calculate the function:
%       r = x^3 - 2*x - 5
% x can be a vector

r = x.^3 - x.*2 -5;
```

% on column 1 is a comment

This is how plot on p.2-7 was obtained

```
» X = 0:0.05:3;
» y = myfunct (x);
» plot(x,y)
```

Tips

- Ctrl + I
  - Auto-align
- Ctrl + R / Ctrl + T
  - Comment / Uncomment
- F9
  - Run selected text