Warmup: Does this script print anything?


```
k= 0;
while 1 + 1/2^k > 1
    k= k + 1;
end
disp(k)
```

A: No – the loop guard is always true

B: Yes, 1/2^k will underflow to 0

C: Yes, 1+1/2^k will round down to 1

D: No – a floating-point error will stop the program

The loop DOES terminate given the limitations of floating point arithmetic!

```
k = 0;
while 1 + 1/2^k > 1
    k = k+1;
end
disp(k)
```

1 + 1/2^53 is calculated to be just 1, so "53" is printed.

Computer arithmetic is *inexact*

- There is error in computer arithmetic—floating point arithmetic—due to limitation in "hardware." Computer memory is finite.

- What is $1 + 10^{-16}$?
  - 1.0000000000000001 in real arithmetic
  - 1 in floating point arithmetic (IEEE double)

- Read Sec 4.3

- Previous Lecture:
  - Discrete vs. continuous; finite vs. infinite
  - Linear interpolation
  - RGB color
  - Floating-point arithmetic
  - Introduction to vectorized computation

lots of new topics!

- Today's Lecture:
  - Vectorized operations
  - Introduction to 2-d array—matrix

$$\begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix}$$

- Announcements:
  - Survey season!
    - Please fill out "Mid-Semester Survey" on CMS
    - Please respond to ENG eval requests (course, TAs, etc.)
  - See website for review materials. Optional review session on Saturday, 11:00am-12:30pm on Zoom
  - Prelim 1 Tuesday 3/30 at 6:30pm, Barton Hall
    - Check CMS for seat!
    - Online practice session: tomorrow evening

Initialize arrays if dimensions are known ("pre-allocation")

… instead of "building" the array one component at a time

```
% Initialize y
x= linspace(a,b,n);
y= zeros(1,n);
for k = 1:n
   y(k)= myF(x(k));
end
```

Faster for large n
BUT you need to know n

```
% Build y on the fly
x=linspace(a,b,n);
for k = 1:n
   y(k)= myF(x(k));
   % OR
   %y= [y myF(x(k));
end
```

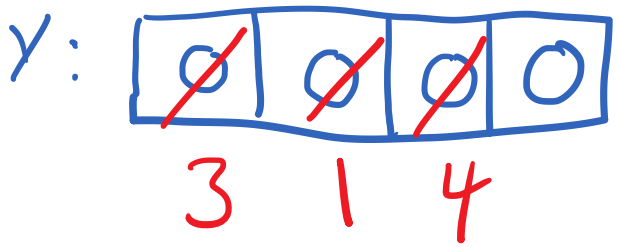
Totally fine if you don't know 'n' (ignore Matlab's warning)

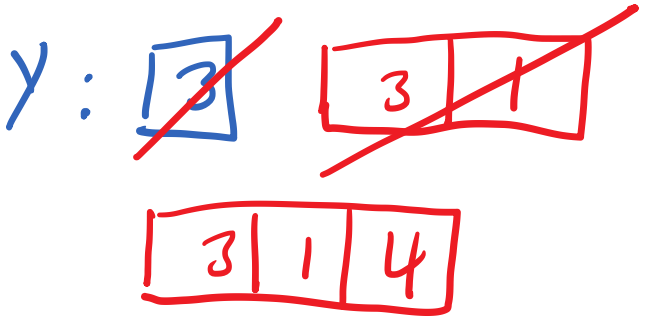
Initialize arrays if dimensions are known ("pre-allocation")

… instead of "building" the array one component at a time

```
% Initialize y
x= linspace(a,b,n);
y= zeros(1,n);
for k = 1:n
   y(k)= myF(x(k));
end
```

```
% Build y on the fly
x=linspace(a,b,n);
for k = 1:n
   y(k)= myF(x(k));
   % OR
   %y= [y myF(x(k));
end
```





Concatenation

- Concatenate two scalars into a (row-)vector:
  `u= [3 1]`
- Concatenate a scalar onto a (row-)vector:
  `v= [u 4]  % v = [3 1 4]`

- Application: repeat the first element of a vector at its end:
  `w= [v v(1)]  % w = [3 1 4 3]`

- Application: append to a vector:
  `w= [w 5]  % w = [3 1 4 3 5]`

Vectorized code

—a Matlab-specific feature

See Sec 4.1 for list of vectorized arithmetic operations

- Code that performs element-by-element arithmetic/relational/logical operations on array operands in one step

- Scalar operation: x + y

  where x, y are scalar variables

  Single value (not containing multiple elements)

- Vectorized code: **x + y**

  where **x** and/or **y** are vectors. Generally, vectors **x** and **y** should have the same length and shape

  rows/columns

Vectorized addition

		1	2	... k	...
	x	2	1	.5	9
+	**y**	1	2	0	2
=	**z**	3	3	.5	11

Matlab code: **z= x + y**

Observe:

$z(1) = x(1) + y(1)$

$z(2) = x(2) + y(2)$

$\vdots$

$z(k) = x(k) + y(k)$

Vectorized multiplication (vector-vector)

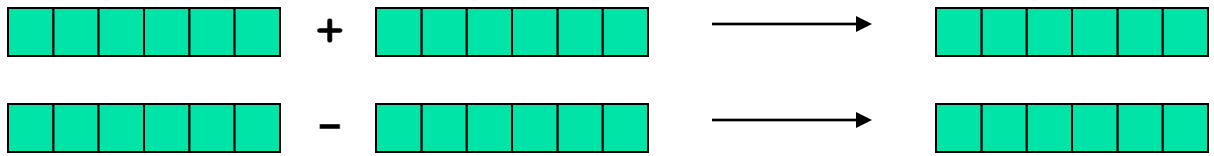
	a	2	1	.5	9
×	**b**	1	2	0	2
=	**c**	2	2	0	18

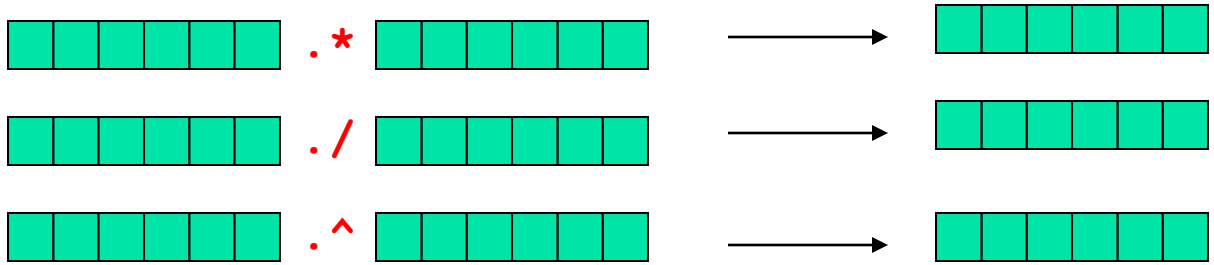
Matlab code: `c= a .* b`

See full list of ops in §4.1

Vectorized

element-by-element arithmetic operations on arrays





A dot (.) is necessary in front of these math operators

Shift (scalar-vector addition)

	x	3			
+	**y**	2	1	.5	9
=	**z**	5	4	3.5	12

Matlab code: `z= x + y`

Reciprocate (scalar-vector division)

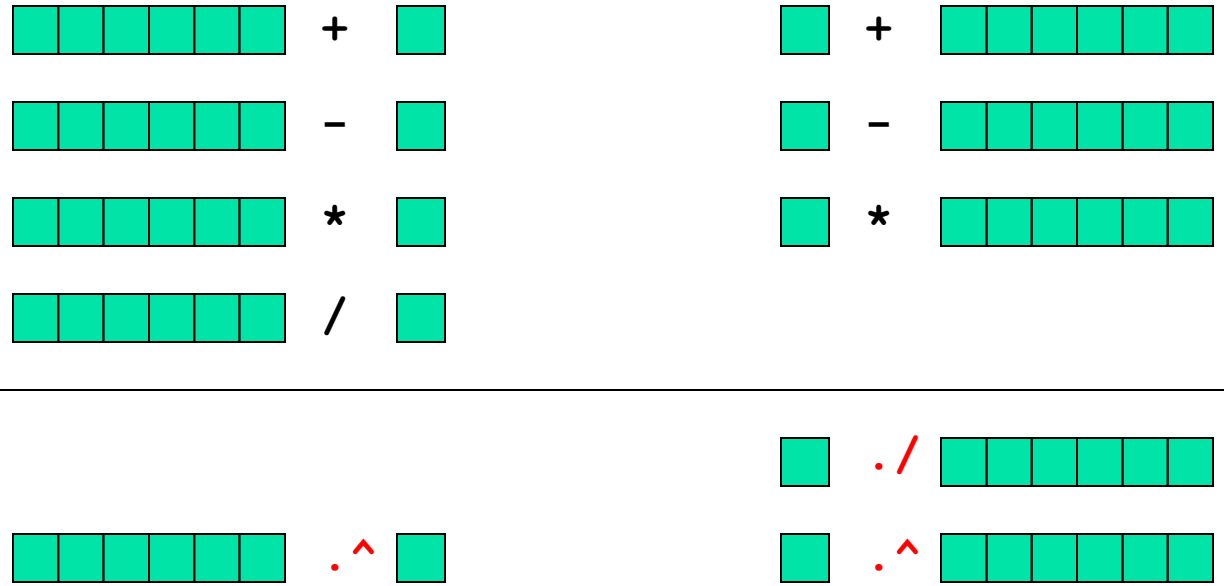
	x	1			
/	**y**	2	1	.5	8
=	**z**	.5	1	2	.125

Matlab code: `z= x ./ y`

See full list of ops in §4.1

Vectorized

element-by-element arithmetic operations between an array and a scalar



A dot (.) is necessary in front of these math operators

Simplified rule: Use dot for these element-by-element ops: * / ^

When are functions vectorized?

- Many built-in functions (`sin()`, `abs()`, …)
- When you only use vectorized operations to implement it
- When you loop over the length of the input
  - Note: Matlab treats scalars like length-1 vectors

```
x = 3.1;
length(x) == 1
x(1) == 3.1
```

Not all functions make sense to vectorize (users can always write their own loops, after all)

See `plotComparison3.m`

Can we plot this?

$$f(x) = \frac{\sin(5x)\exp(-x/2)}{1+x^2}$$

for
-2 <= x <= 3

Yes!

```
x = linspace(-2,3,200);
y = sin(5*x).*exp(-x/2)./(1 + x.^2);
plot(x,y)
```

Element-by-element arithmetic operations on arrays

Element-by-element arithmetic operations on arrays…
Also called "vectorized code"

x and y are vectors

```
x = linspace(-2,3,200);
y = sin(5*x).*exp(-x/2)./(1 + x.^2);
```

Contrast with scalar operations that we've used previously…

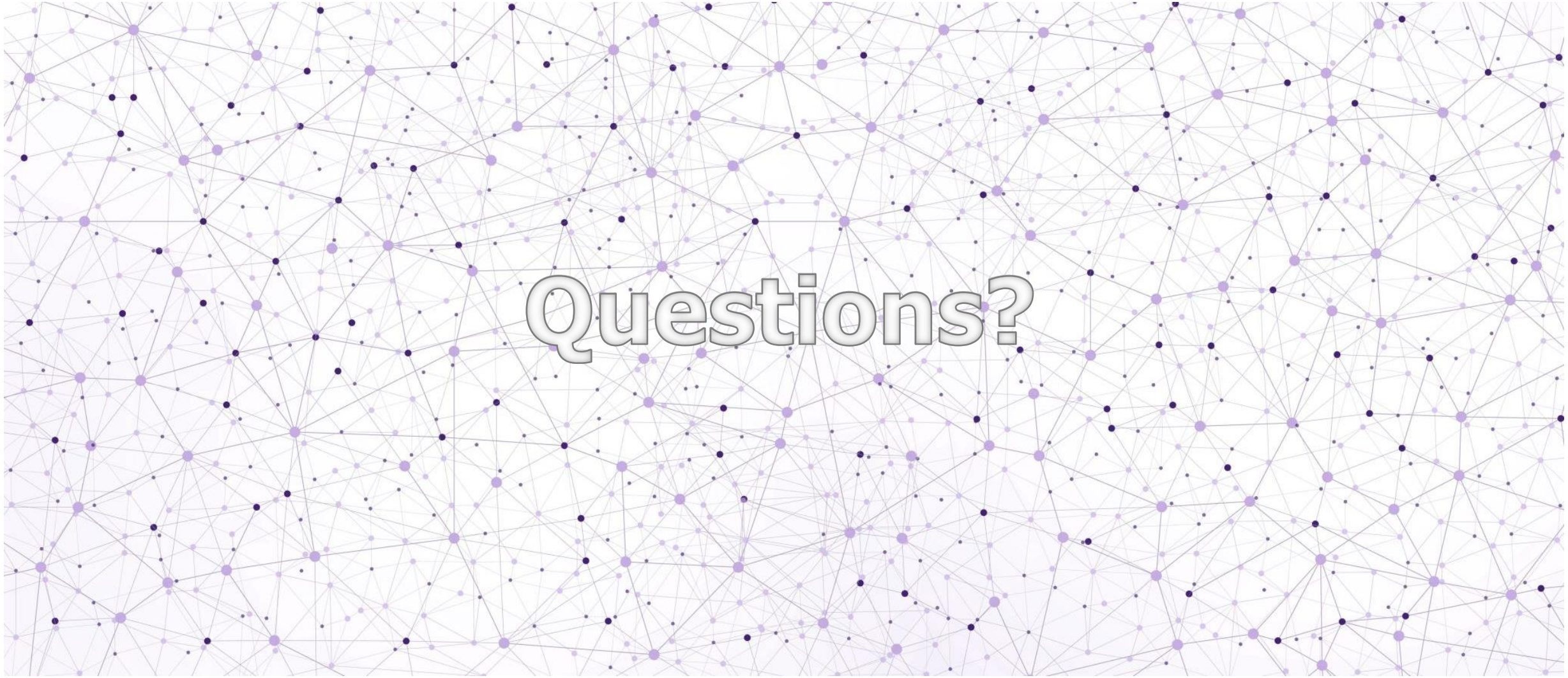
```
a = 2.1;
b = sin(5*a);
```

a and b are scalars

The operators are (mostly) the same; the operands may be scalars or vectors.

When an operand is a vector, you have "vectorized code."

Break



Storing and using data in *tables*

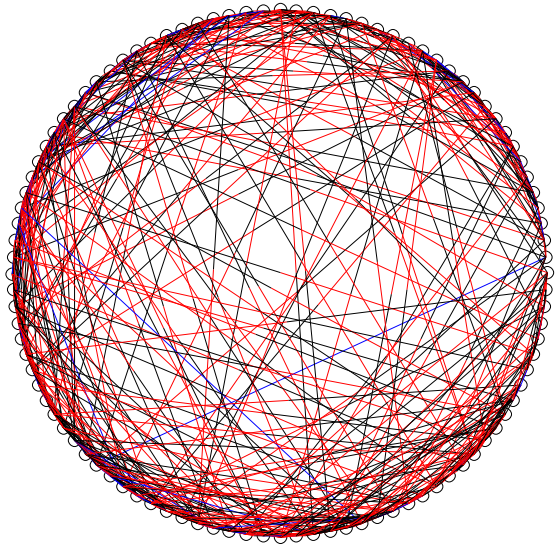
A company has 3 factories that make 5 products with these costs:

Products

factories

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

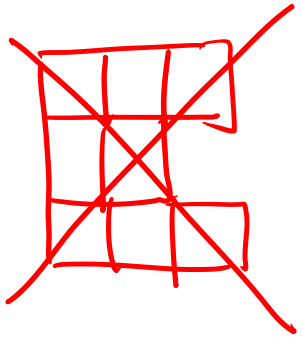
What is the best way to fill a given purchase order?

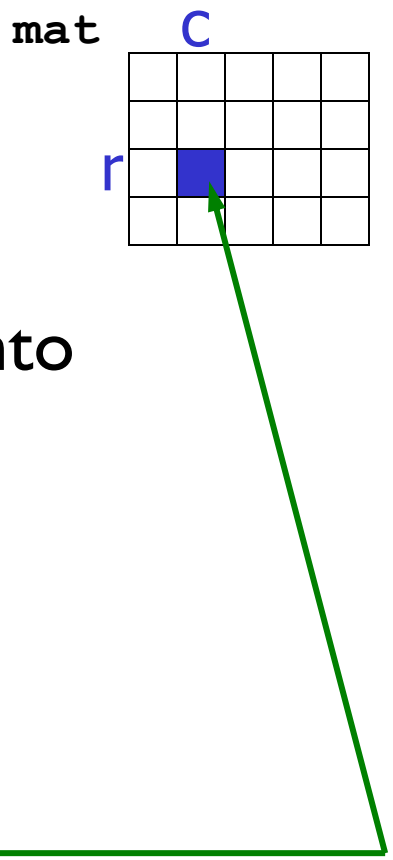


Connections between webpages

0	0	1	0	1	0	0
1	0	0	1	1	1	0
0	1	0	1	1	1	1
1	0	1	1	0	1	0
0	0	1	1	0	1	1
0	0	1	0	1	0	1
0	1	1	0	1	1	0

2-d array: **matrix**





- An array is a named collection of like data organized into rows and columns
- A 2-d array is a table, called a *matrix*
- Two *indices* identify the position of a value in a matrix, e.g.,

  `mat(r,c)`

  refers to component in row r, column c of matrix mat
- Array indices still start at 1
- Rectangular: all rows have the same #of columns

Indexing example

M

refers to the whole matrix

M(1,1)	M(1,2)	M(1,3)	M(1,4)
M(2,1)	M(2,2)	M(2,3)	M(2,4)
M(3,1)	M(3,2)	M(3,3)	M(3,4)

refers to the element (value) in the third row, second column

Creating a matrix

- Built-in functions: `ones()`, `zeros()`, `rand()`
  - E.g., `zeros(2,3)` gives a 2-by-3 matrix of 0s
  - E.g., `zeros(2)` gives a 2-by-2 matrix of 0s
- "Build" a matrix using square brackets, `[ ]`, but the dimension must match up:
  - `[x  y]` puts `y` to the right of `x`
  - `[x; y]` puts `y` below `x`
  - `[4 0 3; 5 1 9]` creates the matrix
  - `[4 0 3; ones(1,3)]` gives
  - `[4 0 3; ones(3,1)]` doesn't work

4	0	3
5	1	9

4	0	3
1	1	1

$$\begin{bmatrix} 4 & 0 & 3 \\ \vdots & ? & \\ \vdots & 0 & \end{bmatrix}$$

Working with a matrix:
`size()` and individual components

Given a matrix M

2	-1	.5	0	-3
3	8	6	~~7~~ 1	7
5	-3	8.5	9	10
52	81	.5	7	2

```
[nr, nc]= size(M)    % nr is #of rows
                     % nc is #of columns
nr= size(M, 1)    % #of rows
nc= size(M, 2)    % #of columns
n= size(M)
```

'size()' is weird like this

```
   % n is length 2 vector since M is 2-d:
   %   n(1) is #of rows, n(2) is #of cols
M(2,4)= 1;
disp(M(3,1))
```

Working with a matrix:
`size()` and individual components

M

2	-1	.5	0	~~7~~ 4
3	8	6	.7	-2
5	-3	8.5	9	10
52	81	.5	1	-8

Given a matrix **M** and the script below

Which statement(s) could make the update shown in **purple** on the diagram?

```
[nr, nc]= size(M);
n= size(M);
M(1,nc)= 4;
M(1,n(2))= 4;
M(0,4)= 4;
```

← A

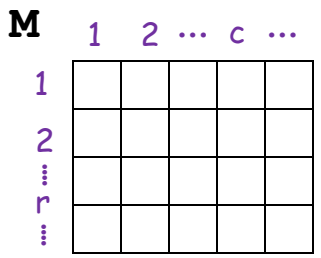
← B

← C

D: None of A, B, C

E: More than one of A, B, C

Example: minimum value in a matrix

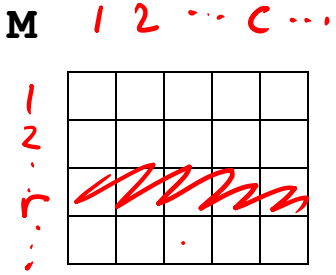



```
function val = minInMatrix(M)
% val is the smallest value in matrix M
```

Best-in-set pattern:

- Initialize best-so-far
- loop over set:
  - If current value is better than best-so-far:
    - Update best-so-far

Example: minimum value in a matrix



function val = minInMatrix(M)

% val is the smallest value in matrix M

```
[nr, nc] = size(M);
val = M(1,1);
for r = 1:nr
    % At row r
    for c = 1:nc
        % At col c (at row r)
        if M(r,c) < val
            val = M(r,c);
        end
    end
end
```

Pattern for traversing a matrix M

```
[nr, nc] = size(M)
for  r= 1:nr
     % At row r
     for  c= 1:nc
          % At column c (in row r)
          %
          % Do something with M(r,c) …
     end
end
```