Warmup: Does this script print anything?

```matlab
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!

```matlab
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

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

- 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

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

```matlab
% 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
```

Faster for large n
BUT you need to know n

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

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

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

% Build y on the fly
% y= [y myF(x(k));
end
Concatenation

- Concatenate two scalars into a (row-)vector:
  \[ \mathbf{u} = [3 \ 1] \]
- Concatenate a scalar onto a (row-)vector:
  \[ \mathbf{v} = [\mathbf{u} \ 4] \quad \mathbf{v} = [3 \ 1 \ 4] \]
- Application: repeat the first element of a vector at its end:
  \[ \mathbf{w} = [\mathbf{v} \ \mathbf{v}(1)] \quad \mathbf{w} = [3 \ 1 \ 4 \ 3] \]
- Application: append to a vector:
  \[ \mathbf{w} = [\mathbf{w} \ 5] \quad \mathbf{w} = [3 \ 1 \ 4 \ 3 \ 5] \]
Vectorized code — a Matlab-specific feature

- 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

- Vectorized code: $x + y$
  where $x$ and/or $y$ are vectors. Generally, vectors $x$ and $y$ should have the same length and shape

See Sec 4.1 for list of vectorized arithmetic operations
Vectorized addition

Matlab code:  \( z = x + y \)

Observe:

\[
\begin{align*}
z(1) &= x(1) + y(1) \\
z(2) &= x(2) + y(2) \\
&\vdots \\
z(k) &= x(k) + y(k)
\end{align*}
\]
Vectorized multiplication (vector-vector)

Matlab code: \( c = a \times b \)
Vectorized

element-by-element arithmetic operations

on arrays

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

See full list of ops in §4.1
Shift (scalar-vector addition)

Matlab code: \( z = x + y \)
Reciprocate (scalar-vector division)

Matlab code: 
```
z = x ./ y
```
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 (\(\sin()\), \(\text{abs}()\), …)
- 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
    
    \[
    x = 3.1; \\
    \text{length}(x) = 1 \\
    x(1) = 3.1
    \]

Not all functions make sense to vectorize (users can always write their own loops, after all)
Can we plot this?

\[ f(x) = \frac{\sin(5x) \exp(-x/2)}{1 + x^2} \quad \text{for} \quad -2 \leq x \leq 3 \]

Yes!

```matlab
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”

```matlab
x = linspace(-2,3,200);     \textbf{x and y are vectors}
y = sin(5*x).*exp(-x/2)./(1 + x.^2);
```

Contrast with scalar operations that we’ve used previously…

```matlab
a = 2.1;                        \textbf{a and b are scalars}
b = sin(5*a);                  \textbf{The operators are (mostly) the same; the operands may be scalars or vectors.}
```

When an operand is a vector, you have “vectorized code.”
Break

Questions?
Storing and using data in **tables**

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

<table>
<thead>
<tr>
<th>Products</th>
<th>10</th>
<th>36</th>
<th>22</th>
<th>15</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>factories</td>
<td>12</td>
<td>35</td>
<td>20</td>
<td>12</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>37</td>
<td>21</td>
<td>16</td>
<td>59</td>
</tr>
</tbody>
</table>

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

  \[ \text{mat}(r,c) \]

  refers to component in row \( r \), column \( c \) of matrix \( \text{mat} \)
- Array indices still start at 1
- **Rectangular**: all rows have the same #of columns
Indexing example

<table>
<thead>
<tr>
<th></th>
<th>M(1,1)</th>
<th>M(1,2)</th>
<th>M(1,3)</th>
<th>M(1,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>M(2,1)</td>
<td>M(2,2)</td>
<td>M(2,3)</td>
<td>M(2,4)</td>
</tr>
<tr>
<td>M</td>
<td>M(3,1)</td>
<td><strong>M(3,2)</strong></td>
<td>M(3,3)</td>
<td>M(3,4)</td>
</tr>
</tbody>
</table>

- **M** refers to the whole matrix.
- **M(3,2)** 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
Working with a matrix:

`size()` and individual components

```
[ 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: 
\textbf{size()} and individual components

Given a matrix $M$ and the script below
Which statement(s) could make the update shown in \textbf{purple} on the diagram?

$$\begin{align*}
[nr, nc] &= \text{size}(M); \\
n &= \text{size}(M); \\
M(1, nc) &= 4; & \leftarrow A \\
M(1, n(2)) &= 4; & \leftarrow B \\
M(0, 4) &= 4; & \leftarrow C
\end{align*}$$

D: None of A, B, C
E: More than one of A, B, C
Example: minimum value in a matrix

```matlab
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 $M$

\[
[nr, nc] = \text{size}(M)
\]

\begin{verbatim}
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
\end{verbatim}