Vectorization: An Introduction

Dr. Marco A. Arocha

INGE3016-MATLAB

Summer 2015

Array and Matrix Operations

Operation	Operator	Explanation
Array addition	a + b	array addition and matrix addition are identical
Array subtraction	a - b	array subtraction and matrix subtraction are identical
Array multiplication	a .* b	element-by-element multiplication of a and b; both arrays must be the same shape, or one of them must be a scalar
Array right division	a ./ b	element-by-element division of a by b: a(i,j)/b(i,j); both arrays must be the same shape, or one of them must be a scalar
Array left division	a .\ b	element-by-element division of b by a: b(i,j)/a(i,j); both arrays must be the same shape, or one of them must be a scalar
Array exponentiation	a .^ b	e-by-e exponentiation of a to b exponents: a(i,j)^b(i,j); both arrays must be the same shape, or one of them must be a scalar
Matrix Multiplication	a * b	the number of columns in a must equal the number of rows in b
Matrix right division	a / b	a * inv(b), where inv(b) is the inverse of matrix b
Matrix left division	a \ b	inv(a) * b, where inv(a) is the inverse of matrix a
Matrix exponentiation	a^b	matrix multiplication of a: a*a*a*...a, b times

Vectorization

- The term “vectorization” is frequently associated with MATLAB.
- Means to rewrite code so that, instead of using a loop iterating over each scalar-element in an array, one takes advantage of MATLAB’s array operators and does everything in one go.

- It is equivalent to change a Yaris for a Ferrari

Why Vectorization?

Matlab is optimized for operations involving matrices and vectors. The process of revising loop-based, scalar-oriented code to use matlab matrix, array and vector operations is called vectorization. Vectorizing your code is worthwhile for several reasons:

- Appearance: Vectorized mathematical code appears more like mathematical expressions, making the code easier to understand.
- Less Error Prone: Without loops, vectorized code is often shorter. Fewer lines of code mean fewer opportunities to introduce programming errors.
- Performance: Vectorized code runs much faster than the corresponding code containing loops.

Vectorization

Operations executed one by one

(NO vectorized):

```
x = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];

for ii = 1:1:numel(x)
        y(ii) = x(ii)^3;
end
```

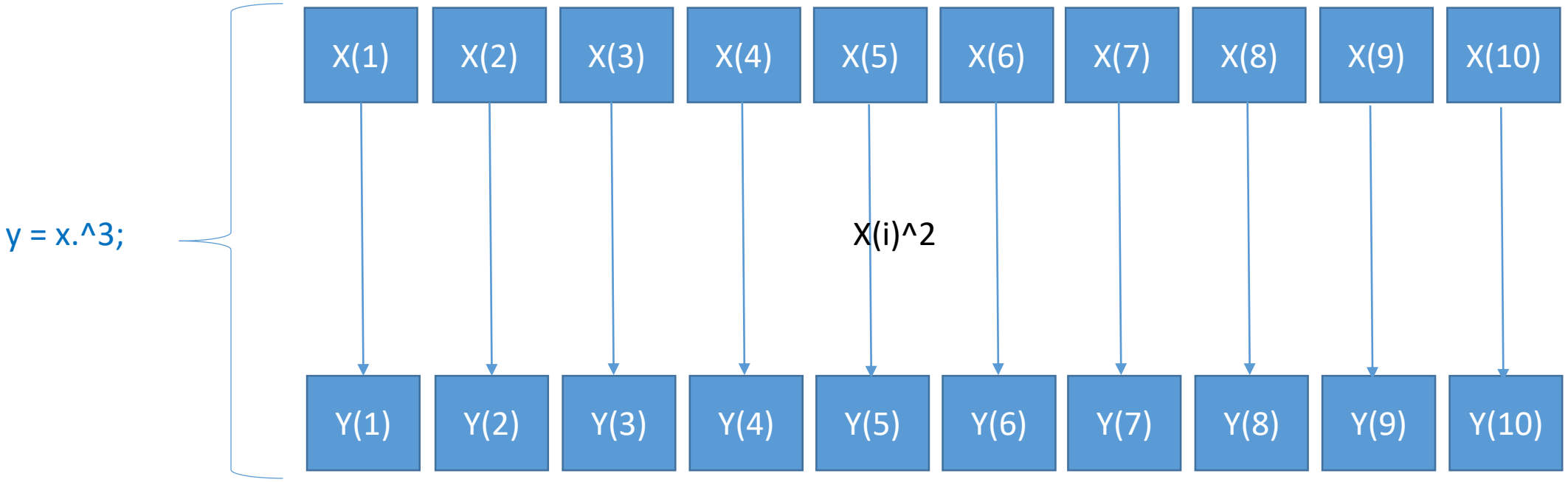
```
% Vectorized code:

x = [ 1 :1:10 ];
y = x.^3;
```

Exercise:

Rise to the 3rd power only the first five elements of x:

```
x=[1:1:10];
ii=[1:1:5];
y(ii)=x(ii).^3;
```

Operations happens in parallel (all at the same time), not in series



Vectorization

Operations executed one by one:

```
x = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];

for ii = 1:1:numel(x)
        y(ii) = sin(x(ii))/x(ii);
end
```

Vectorized code:

```
x = [ 1 :1:10 ];
y = sin(x)./x;

% MOST matlab functions are prepared to work with array arguments in element-wise fashion
% Answer the following questions:  How many elements does the x-array have?, sin(x)-array?, y-array?
```

Vectorization

Operations executed one by one:

```
A=rand(100,1);
B=rand(100,1);
for ii=1:1:100
        C(ii)=A(ii)*B(ii);
end
```

Vectorized code:

```
A=rand(100,1);
B=rand(100,1);
C=A.*B;
```

Vectorization

Operations one by one:

```
A=rand(100,1);
B=rand(100,1);
for i=1:100
  if B(i)>0.5
    C(i)=A(i)^2;
  else
    C(i)=exp(B(i));
  end
end
```

Vectorized code:

```
A=rand(100,1);
B=rand(100,1);
D=(B>0.5);
C=D.*(A.^2)+(~D).*exp(B);
```

Here D is a logical array of 0 and 1 of the same size as A and B. For each element in B that is superior to 0.5, there is a 1 or true in the corresponding element in D (and vice versa).

"~" is the "logical not" operator so "~D" is the exact opposite of D. Ones are now zeros and vice versa.

C(i) elements: If elements of B(i) are greater than 0.5, compute $A(i)^2$ otherwise compute $e^{B(i)}$

Vectorization

Operations executed one by one:

```
% 10th Fibonacci number (n=10)
n=10;
f(1)=0;
f(2)=1;

for k = 3:1:n
        f(k) = f(k-1)+f(k-2);
end
```

WRONG Vectorization:

```
% 10th Fibonacci number (n=10)
n=10;
f(1)=0;
f(2)=1;

k= [ 3 :1:n];
f(k) = f(k-1)+f(k-2);
```

CAN'T

All elements in the RHS of
f(k) = f(k-1)+f(k-2);
must exist in memory, but
they don't

Vectorization

Operations executed one by one:

```
% Find factorial of 5:  5!
x=[1:1:5];
f=1;
for ii = 1:1:length(x)
        f=f*x(ii);
end
```

Wrong Vectorization: Why this code doesn't work?:

```
x=[1:1:5];
f(1)=1;

ii=2:1:length(x);
f(ii)=f(ii-1)*x(ii);
```

CAN'T

All elements in the RHS of
f(ii) = f(ii-1)*x(ii);
must exist in memory. They don't

Vectorization-Exercise:

Vectorize the following loop:

```
for ii=2:1:n-1
   tn(ii)=(to(ii-1)+to(ii+1))/2;
end
```

Note: “to”, the old temperatures array has been initialized previously, i.e., all elements already exist in memory

Note: the calculated values are only the interior nodes

Vectorized code:

```
ii=[2:1:n-1];
tn(ii)=(to(ii-1)+to(ii+1))/2;
```

Vectorize

$$vector = 1^2, 2^2, 3^2, 4^2, 5^2, \ldots, 100^2$$

Operations executed one by one

(NO vectorized):

```
clc,clear
kmax = 100;
k = 1;
while k <= kmax
    vector(k) = k^2;
    k = k + 1;
end
```

```
% Vectorized code:
clc, clear
kmax=100;
k=1:1:kmax;
vector(k)=k.^2;
```

Also:

```
% Vectorized code:
clc, clear
kmax=100;
k=1:1:kmax;
vector=k.^2;
```

Vectorization Summary

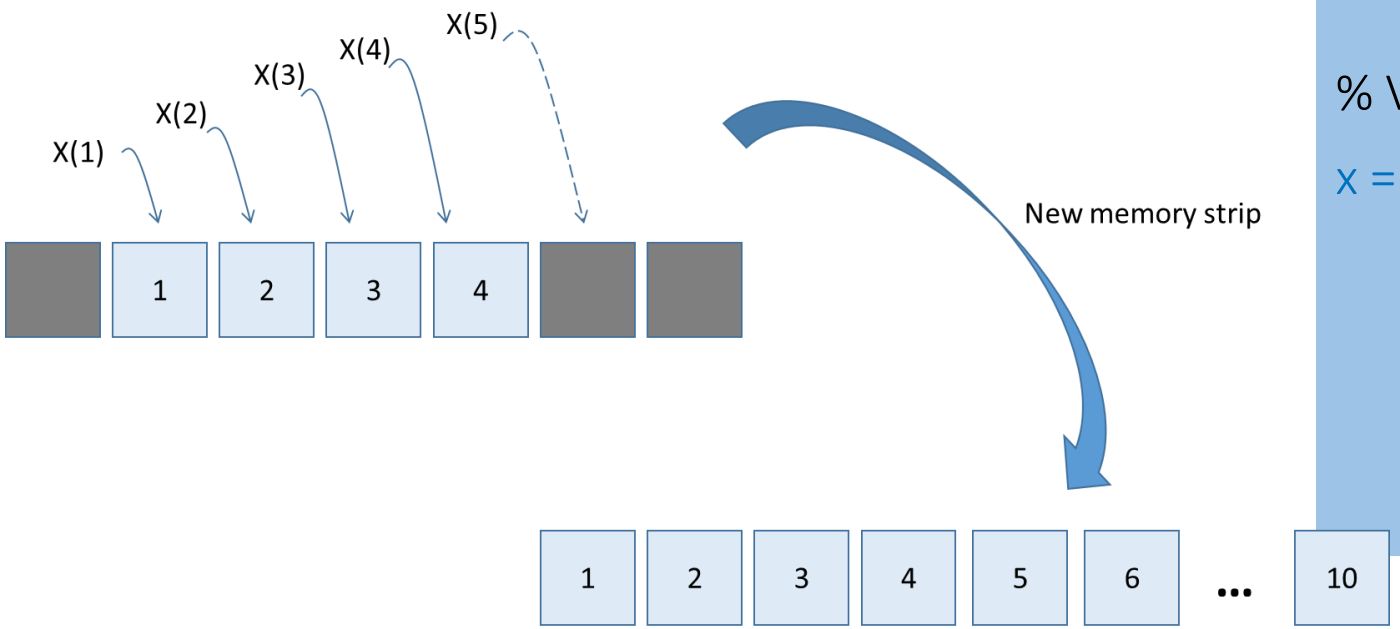
- ✓Colon operator : (or linspace function)
- ✓Array Operators: .* .\ ./ .^ + -
- ✓Matrix Operators: * \ / ^ + -
- ✓Library Functions: (where the input argument is an array/matrix/vector)
- ✓Omit loops and if statements (when possible)
- ✓Not all program codes can be ‘fully’ vectorized.
- ✓VECTORIZED code is faster (worthwhile the effort)

memory pre-allocation

In Matlab, arrays are dynamic in size, meaning, whenever you need a larger array, you can make it bigger. There is a cost with doing this, which is that your program will slow down. Thus, if you know (or have a good guess) at how big the array needs to be in the first place, you can "pre-allocate" it, meaning pre-size the array; reserve space in memory to handle the array.

Memory Fragmentation

```
for i=1:10
    x(i)=i;
end
```




```
% Memory Pre-Allocation
x=zeros(1,10);   % to speed code

% Vectorized code:
x = [ 1 :1:10 ];
```

The first statement pre-allocate the memory that x-arrays will use in the program. This makes the codes to run faster

At the i=1, ML requests enough memory from the OS to create a 1 x 1 matrix, and creates x(1)=1. Next i=2, ML requests more memory so a 1 x 2 matrix can be stored. If this additional memory is in the same continuous memory strip as when x(1)=1, ML will simply add the additional number in the same memory strip, and so on. If the original memory strip is only big enough for a 1x4 matrix, ML moves the x(1),x(2),x(3),x(4) trip and places it into a memory spot that is large enough for the 1x5 matrix. Since the matrix is now 1x5, the original memory slot is useless to ML for any matrix larger than 1x4. Memory is now fragmented, and this would cause significant problems with large FOR loops.

Memory Pre-Allocation &Vectorization

Operations executed one by one (NO vectorized) with memory pre-allocation:

```
% Memory Pre-Allocation
x=zeros(1,10);
y=zeros(size(x));
x = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
for k = 1:1:numel(x)
        y(k) = x(k)^3;
end
```

```
% Memory Pre-Allocation
x=zeros(1,10);
y=zeros(size(x));   } to speed code

% Vectorized code:
x = [ 1 :1:10 ];
y = x.^3;
```

The first two statements (in both codes) pre-allocate the memory that x- and y-arrays will use in the program. This makes the codes to run faster

Memory Pre-Allocation &Vectorization

```
% Memory Pre-Allocation
x=zeros(1,10);
y=zeros(size(x));        % to speed code


% Vectorized code:
x = [ 1 :1:10 ];
y = x.^3;                % These two strategies (red and
                         % blue) will make the whole code to
                         % run faster
```

Read documents:

- preallocate MEMORY.pdf
- Preallocation is not an option.pdf

Measure Execution Performance

tic and toc, enable you to time how long your code takes to run.

```
clc, clear

tic
for i=1:10
    x(i)=i;
end
toc
```

OUTPUT

Elapsed time is 0.002114 seconds.

```
clc, clear

tic
x=zeros(1,10);  % Pre-Allocation
x = [ 1:1:10 ];    % Vectorized
toc
```

OUTPUT

Elapsed time is 0.000615 seconds.

How many times faster?