Lecture 3: MATLAB — advanced use cases

Efficient programming in MATLAB

Juha Kuortti and Heikki Apiola

February 17, 2018

Aalto University
juha.kuortti@aalto.fi

Before we begin

On this lecture, we will mostly be talking about efficiency, and more specifically about the time it takes to run the programs. In MATLAB, we time the programs using commands `tic` and `toc`; `tic` starts a stopwatch, and `toc` tells how many seconds has passed since the last `tic`.

For more advanced and detailed reports on the time taken, you can use `profile`.

Preallocation

The origin of the issue

MATLAB is an array based language, and therefore it has some built-in compromises when it comes to memory allocation. Consider the following:

```
a = 0;
a(2) = 1;
a(3) = 3;
```

Most other languages would consider this an assignment out of range, but in MATLAB, as long as the dimensions remain unambigous, the assignment out of range is valid code.

Assignment out of index range

Assigning out of index range can be incredibly useful. Consider reading in some unknown amount of data. Consider

```
f = @(x)x.^3-3*x+5;
k = 1;
while f(x)<0
  y(k) = f(x);
  k = k+1;
  x = x+0.1;
end
```

There is a price to be paid for doing things easy

Run these two code segments, and report back the times it takes to run them.

Segment 1:

```
tic;
N = 10000;
x(1) = 1000;
for k = 2:N
   x(k) = 1.05*x(k-1);
end
toc;
```

Segment 2:

```
tic;
N = 10000;
x = zeros(N,1);
x(1) = 1000;
for k = 2:N
   x(k) = 1.05*x(k-1);
end
toc;
```

Exercise I

Load the file `blocAvg_bad.m` from the course resource page. The purpose of the function is to downsample the surface created by function $f(x, y) = 5\cos(2(x + y)\pi) + 2\sin(2x\pi) + 2\cos(2x\pi)$.

At its current form the function works correctly, but has some efficiency problems. Use `tic` and `toc` to observe the effects of preallocation.

Also, instead of preallocation, see what happens if you try to run the loops in reverse order (i.e. using increment of -1). Is there speedup?

Loops and indices

Vectorization

In the course of computer history *vectorization* has had a lot interpretations. Usually, regardless of context it means doing many things simultaneously. In the context of MATLAB, it means taking full advantage of vectorized functions and operators. Consider

```
N = 512;
x = zeros(N,1); y = x;
for k = 2:N
   x(k) = x(k-1) + ...
       k*pi/N;
end
for k = 1:n
   y(k) = sin(x(k));
end
```

```
x = linspace(0,pi);
y = sin(x);
```

Column majority

In MATLAB, matrices are stored in *contiguous array* in columnwise fashion – in the order that is given by linear indices. This allows us to use memory hierarchy to our advantage — if we can access the contiguous elements of the array, we can see a significant performance boost:

```
A = rand(10000);
tic
for i = 1:N
    for j = 1:N
        if A(j,i)>refNo
           vals(ix)=A(j,i);
           ix = ix+1;
        end
    end
end
toc
```

```
A = rand(10000);
tic
for i = 1:N
    for j = 1:N
        if A(i,j)>refNo
           vals(ix)=A(i,j);
           ix = ix+1;
        end
    end
end
toc
```

Subscript and linear indices

Previously we used so called *subscript indexing* when traversing the elements of the matrix. However, as stated, MATLAB does not store matrices as "arrays of arrays," but rather as a single contiguous array, meaning every subscript indexing also involves a computation. Let's compare performance with using linear indices only.

```
tic
for i = 1:N^2
    if A(i)>refNo
       vals(ix)=A(i);
       ix = ix+1;
    end
end
toc
```

Exercise

Let's go back to the `blockAvg_bad.m`. Previously we managed to speed it up quite a bit by doing preallocation. This time we'll seek additional speedups: what happens if you switch the loop orders?

How about linear indices? Can you use them? If so, is there speedup?

For demo purposes, we'll also take a look at vectorisation.

Logical indexing

Logical indexing offers a syntactically great way to do searches and handle indexing. It is also very fast: let't observe the previous examples with logical indexing.

```
tic
vals = A(A>refNo);
toc
```

Note: Logical indexing usually involves an implicit search operation. If you have pure indices, then (usually) using linear indices will be faster.

Set operations

MATLAB contains the standard set operations : union, intersect, difference etc. These have their uses, especially when dealing with more complicated datastructures, but on numerical data, it usually suffices to use the logical equivalents of the set operations.

Demo about binning

arrayfun, cellfun, bsxfun – what's with all the fun

Applying function to arrays I

Suppose you have a huge number of processors at you disposal — say, a GPU with CUDA enabled or a Xeon Phi, or similar; so many that technically you could assign one of them for every element of you array. In situation like that you might want a way to *apply* some function to every element of an array.

Most of the elementary functions already automatically traverse arrays, but more complicated ones might not; and most do not traverse cell arrays.

To this end MATLAB implements a functions called `arrayfun` and `cellfun`

Extended matrix operations

Suppose you have a function that you want to apply to every *row* or *column* of a matrix.

Before version 2016b, you had to use a function called `bsxfun`. In 2016, MATLAB basic operators were updated to use extended operations — nowadays, most operations are supported automatically.

Try out the following commands, and see what they do.

```
A = magic(6); b = (1:6)'; c = (1:7);
A+b
A+c
b+c
c.*b
c*b
```