Principles of Technical Computing

Lecture 5 – Speed, style, trickery, and other useful stuff

Matylda Jabłońska-Sabuka

Lappeenranta University of Technology

Overview of Week 6

- Memory preallocation
- Vectorization
- Masking (logical indexing)
- NaN (not a number)
- is*something* functions

Memory preallocation

Memory preallocation

Most of operations in Matlab can be executed even without preallocating sufficient memory. That is, we do not have to declare how big our output vector/matrix/variable will be.

However, smart preallocation of memory in cases when it is possible, is a time-saving effort.

Example – code with no preallocation

```
n = 10^8;
y = rand(1,n);
for i = 1:n
    z(i) = y(i)*2;
end
```

This code takes about 11.57s (414) to run, because every time Matlab creates the new `i`-th element of `z`, it has to check the size of `z` first.

Example – code with preallocation

```
n = 10^8;
y = rand(1,n);
z = ones(1,n);
for i = 1:n
    z(i) = y(i)*2;
end
```

This code takes about 3.41s (159) to run, because Matlab knew from the very beginning how much space is reserved for z.

Vectorization

Vectorization

Vectorization refers to the removal of loops (`for` and `while`).

When you need to operate on big vectors or matrices, vectorization can reduce computational time. For instance in the previous example the the vectorized form z = y.*2 needed only 2.08s (64).

As another example, suppose x is a column vector and you want to compute a matrix D such that $d_{ij} = x_i - x_j$.

Example – nested loops

```
x = rand(10000,1);
n = length(x);
D = zeros(n); % preallocation
for j = 1:n
    for i = 1:n
        D(i,j) = x(i) - x(j);
    end
end
```

This code takes about 1.05s to run.

Example – one loop vectorized

```
x = rand(10000,1);
n = length(x);
D = zeros(n); % preallocation
for j = 1:n
    D(:,j) = x - x(j);
end
```

This code takes about 0.47s to run.

Example – full vectorization

```
x = rand(10000,1);
n = length(x);
X = repmat(x,1,10000);
D = X - X';
```

This code takes about 1.25s to run.

Why vectorize?

There are two answers: speed and style
(though style is a subjective matter, of course).

Not every loop can be optimized due to changing acceleration techniques. Therefore, code vectorization is no longer speed-critical in every case.

In our example a medium level of vectorization proved fastest.

Masking (logical indexing)

Masking

An advanced type of vectorization is called masking.

Let's say that we have a vector x of values at which we want to evaluate the piecewise-defined function

$$f(x) = \begin{cases} 1 + \cos(2\pi x), & |x| \leq \frac{1}{2}; \\ 0, & |x| > \frac{1}{2}. \end{cases}$$

Example – standard loop method

```
x = rand(10000,1);
f = zeros(size(x));
for j = 1:length(x)
    if abs(x(j)) <= 0.5
        f(j) = 1 + cos(2*pi*x(j));
    end
end
```

This code runs in about 0.082s.

Example – shorter way using a mask

```
x = rand(10000,1);
f = zeros(size(x));
mask = (abs(x) < 0.5);
f(mask) = 1 + cos(2*pi*x(mask));
```

This code runs in about 0.07s.

The mask is a logical index into `x`. You could refer, if needed, to the unmasked points by using `~mask`.

NaN (not a number) in Matlab

Not a number (`NaN`)

`NaN` is used to express an undefined value.
For example, `Inf/Inf = NaN`.

`NaN`s can also appear when we load data with missing values.

`NaN`s can be tricky. For example, any mathematical operations on a set of numbers and at least one not a number, will return a `NaN`.

One of the few Matlab functionalities which smartly omit `NaN`s are most plots.

`isnan` – verifies whether something is `NaN` or not

`isnan(NaN)` returns 1

`isnan(54)` returns 0

`isnan('text')` returns 0

```
x = [3 5 10 -7 NaN 1 -2];
isnan(x)
```

returns

```
ans =

    [0   0   0   0   1   0   0];
```

Example: We load data from an external file. How do we check for `NaN`s?

```
  6     25
  7    NaN
  8     31
  9     34
 10     37
 11     40
 12     43
 13     46
 14    NaN
 15     52
```

```
isnan(data)

ans =

     0     0
     0     1
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     1
     0     0
```

However, for large data sets looking for 1 in a huge matrix may not be that easy.

How many are there?

If we only need to know how many missing values we have we can check that either by columns:

```
sum(isnan(data))


ans =


     0     2
```

or in total

```
sum(sum(isnan(data)))


ans =


     2
```

Where are they?

If we need to know their location (for instance to replace them):

```
find(isnan(data)==1)  % for single index information


ans =


    12
    19
```

or

```
[r,c] = find(isnan(data)==1); [r c]  % for row and column


ans =


     2     2
     9     2
```

Replacing `NaN`s

For instance, if we want to replace every missing value with a zero:

```
data(isnan(data)==1)=0
```

or

```
data(find(isnan(data)==1))=0
```

Decision whether we should replace the `NaN`s with other values or omit them completely in the calculation, is very subjective based on the problem and data.

If, for instance, you just need to find some statistical features of the data (like mean value), you can easily omit the missing values.

```
mean(x(isnan(x)==0))
```

If the data matrix has two columns, and we need the mean value of just one of the columns (let's say second one), it can be done this way

```
mean(x(isnan(x(:,2))==0,2))
```

is*Something* functions

`ischar` – verifies whether the variable is a string of characters

`ischar('anytext')` returns 1
`ischar(87)` returns 0
`ischar(NaN)` returns 0

`isempty` – checks whether an array has any elements

`isempty([])` returns 1
`isempty('anytext')` returns 0
`isempty([1 1;1 1])` returns 0

`isnumeric` – checks whether an array is numeric

`isnumeric([])` returns 1
`isnumeric([1 3 NaN 8])` returns 1
`isnumeric('anytext')` returns 0

Verification of user input

Such functions can be used when assuring that any user of our script or function will provide a proper input for calculation.

All we need to do is combine them smartly with loop `while` and function `input`.

Example

Write a script which asks for one number as input. Then:

- if the number equals 10, assign $x = 0$
- if the number is greater than 10, assign $x = 1$
- if the number is less than 10, assign $x = -1$.

Display the result in the Command Window.

It may happen that the user, instead of giving a number, will just press enter, type in text, or insert an array instead of a single number.

The following code will prevent such inputs.

```
a = input('Give a number: ')

while isnumeric(a)==0 || isempty(a)==1 || length(a)>1
    a = input('The input is not correct. Try again: ');
end

if a>10
    x=1;
elseif a<10
    x=-1;
else
    x=0;
end
x
```

Similar prevention can be done for other restrictions.

For instance, we write a function which will create first *n* Fibonacci numbers, with *n* given by the user.

We want to prevent *n* from being:

- non-integer (for instance 4.65)
- empty (if the user presses enter)
- negative
- any array bigger that 1x1
- text

```
function F = nFib

n = input('How many Fibonacci numbers do you need? ');
while n~=round(n) || n<0 || isempty(n)==1 ...
        || length(n)>1 || isnumeric(n)==0
    disp('The number has to be a positive integer.')
    n = input('Try again: ');
end
F = zeros(n,1);
if n==1
    F(1) = 1;
elseif n==2
    F(1:2) = [1 1];
else F(1:2) = [1 1];
    for i = 3:n
        F(i) = F(i-1) + F(i-2);
    end
end
```

Write a function which for any given n, representing the tree's total hight, will produce a Christmas tree looking like this (this one has total height 13):

```
         *
        / \
       / | \
      / / \ \
     / / | \ \
    / / / \ \ \
   / / / | \ \ \
  / / / / \ \ \ \
 / / / / | \ \ \ \
/ / / / / \ \ \ \ \
-------------------
         *
         *
```