Writing Efficient MATLAB® Codes

Reza Sameni*
Revision: 5 Aug, 2020

1 Introduction

MATLAB[1] is one of the most practical software used for numerical calculations, signal processing and machine learning. It is a programming language, which has been optimized for matrix computations. The computational core of MATLAB is the LAPACK and BLAS libraries which were originally written for matrix calculations in Fortran. The first versions of MATLAB were just an interface to these libraries. However, it soon evolved into a programming language with many functions and toolboxes for various applications.
In this short tutorial we review some of the general guidelines for writing more efficient MATLAB codes. By efficient we mean more compact code size, better memory access, and shorter execution time. Note that some of these properties are machine dependent concepts. The later presented performance benchmarks have been achieved on a 1.5 GHz Centrino Notebook with 512 MBytes of RAM, and 1 MBytes of Cache memory.

2 General guidelines

General guidelines for improving MATLAB codes:

1. Improve the algorithm before attempting to optimize the code.
2. Benefit from the matrix abilities of MATLAB and its built-in functions.
3. Avoid writing the entire program in a single m-file script. One should break down a program into separate m-file functions. This also will help the reuse of functions and creating (open-source) toolboxes.
4. Comment the codes.
5. MATLAB version 6.5 and later support object-oriented programming. This feature can be used to make more structured and reusable codes.

3 Some of the general features of MATLAB

MATLAB is an 'interpreter' language, which can also use pre-compiled components. This is a good feature for debugging programs and software development. But at the same time it makes MATLAB rather slow (as compared to languages such as C and C++). It is therefore better to use its very efficient components– known as built-in functions– as much as possible, rather than using long scripts with nested for-loops and if-conditions. All the predefined MATLAB constants are in fact built-in functions. For example:

- `pi` is $4 \times \arctan(1)$

*Department of Biomedical Informatics, Emory University School of Medicine, GA 30322, US, Web: `www.sameni.info`, Email: `rsameni@dbmi.emory.edu`
[1]MATLAB stands for 'Matrix Laboratory'.

- `i` and `j` are $\sqrt{-1}$. One can always redefine `i = sqrt(-1)` if it is overloaded by a previous m-file script.
- `eps(x)` and `eps(single(x))` are the positive distance from `abs(x)` to the next larger in magnitude floating point number of the same precision as `x`.
- `realmin` and `realmax` are machine dependent single- or double-precision positive floating point numbers.
- `inf` returns the IEEE arithmetic representation for positive infinity, which is also produced by operations like division by zero, e.g. 1.0/0.0, or from overflow, e.g. exp(1000). It may also be used in matrix form:

```
>> a = inf(2,2)
a =
   Inf   Inf
   Inf   Inf
```

- `NaN` is the IEEE arithmetic representation for not-a-number. A NaN is obtained as a result of mathematically undefined operations like 0.0/0.0 or inf−inf. It may be used in matrix form as follows:

```
>> nan(3)
ans =
   NaN   NaN   NaN
   NaN   NaN   NaN
   NaN   NaN   NaN
```

All of these parameters are in fact built-in functions. One can check whether a variable or function is a built-in or not as follows:

```
>> type i
i is a built-in function.
```

or:

```
>> exist('i')
ans =
     5
```

Built-ins may be overloaded by constants or functions. For example, `i=1` or:

```
function F= fft(x)
    F = x.^2
```

One may clear the over-loaded values to retrieve the original built-in values:

```
>> clear i j
```

It is possible to use recursive functions in MATLAB. For example, the following calculates $n! = 1 \times 2 \times \ldots \times n$:

```
function f = fact(i)
if i < 2
    f = 1;
else
    f = i*fact(i-1);
end
```

4 Memory access

MATLAB can dynamically allocate memory:

```
>> a(1,3) = 5

a =
     0     0     5

>> a(2,4) = 6

a =
     0     0     5     0
     0     0     0     6
```

However one can also pre-allocate the required memory:

```
>> a = zeros(1000,200);
```

which makes the codes run much faster, especially for large arrays. Let us see an example:

```
N = 50000;
a(1) = 1;
a(2) = 2;
t = cputime;        % tic
for i = 3 : N
    a(i) = 0.9*a(i-1) - 0.3*a(i-2);
end
dt1 = cputime - t   % toc

clear a;

a = zeros(1,N);
a(1) = 1;
a(2) = 2;
t = cputime;        % tic
for i = 3 : N
    a(i) = 0.9*a(i-1) - 0.3*a(i-2);
end
dt2 = cputime - t   % toc
dt1/dt2
```

which results in:

```
dt1 = 5.6406
dt2 = 0.0469
dt1/dt2 = 120.3333 !
```

In similar examples – depending on the CPU, RAM, Cache, required memory, and the type of computation– a code may run orders of magnitude faster when one pre-allocates the required memories. Pre-allocation may also be used for variable length vectors which have an upper-bound of required memory:

```
% Note: This code is just for memory allocation illustration, but it is
%  not efficient in speed. You will see the reason in later sections.
N = 1000;
a = rand(N,1);
b = zeros(size(a));
k = 0;
for i = 1 : N
   if(a(i) > 0.5)
      k = k + 1;
      b(k) = a(i);
   end
end
b = b(1 : k);
```

The downside of memory pre-allocation is when huge amounts of memory are required, which may not be allocated in a single memory block. Test the following examples (and change the loop ranges and matrix sizes depending to your system's resources):

```
>> % compare the following codes:
>> A = zeros(5000,10000); % can be too big to be allocated!
>> for i = 1 : 10000       % this one is allocated easier.
       B(:,i) = zeros(5000,1);
   end
```

One can use the function `pack` which performs memory garbage collection by saving the current workspace on the hard disc, clearing the workspace, and reloading the workspace from the hard disc. It is very useful for defragmentation of the memory in long-running programs that continuously allocate and clear variables. Note that it is not efficient to use `pack` so frequently, since it slows down the code. For example:

```
for i = 1 : N
    if (mod(i,100)==0) % perform packing every 100 iterations
        pack;
    end
    f(i);   % a user-defined function which fragments the memory
end
```

5 Speed optimization

MATLAB is excellent in matrix calculations. So it is very important to use its matrix abilities rather than using for-loops or if-conditions, which operate on single entries of vectors and matrices.

5.1 Array indexing

There are three ways of array indexing in MATLAB:

1. *Subscripted*

```
>> A = [11 14 17;...
        12 15 18;...
        13 16 19];
>> A(1,2)

ans =

    14

>> A([1 2],2)

ans =

    14
    15

>> A(1:3,1)'

ans =

    11    12    13

>> A(1:end,2)

ans =

    14
    15
    16
```

2. *Linear*

```
>> A(1)

ans =

    11

>> A(4)

ans =

    14
```

MATLAB matrices are stored in column order (unlike C where matrices are stored in row order). As a result, for an $M \times N$ matrix `A`, subscript and linear indexings can be related to each other as follows:

```
A(i + (j-1)*M) % which is equivalent to A(i,j)
```

and

```
A(rem(index-1,M)+1,floor((index-1)/M)+1) % which is equivalent to A(index)
```

Note that by using the colon operator (`:`) one can convert a multi-dimensional matrix to a column vector (obtained by stacking its columns under one another):

```
>> A(:)
```

This feature can be used to, for example, change the entries of a matrix without reshaping it:

```
>> A(:) = 1;
```

3. *Logical*

`logical` is a data-type in MATLAB, which is returned by many of the comparative operators. This feature is helpful in writing compact and fast codes without using if-conditions. Let us take a look at some examples:

```
>> [4 5 3] > [1 2 6]

ans =

     1     1     0

>> whos
  Name      Size                    Bytes  Class

  ans       1x3                         3  logical array

Grand total is 3 elements using 3 bytes

>> A = 6 : 10;
>> A(logical([0 0 1 0 1]))

ans =

     8    10

>> A(A > 7)

ans =

     8     9    10
```

```
>> B = randn(3)

B =

   -0.0956   -1.3362   -0.6918
   -0.8323    0.7143    0.8580
    0.2944    1.6236    1.2540

>> C = rand(3)

C =

    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

>> C(B > 0)

ans =

    0.6068
    0.8913
    0.7621
    0.0185
    0.8214
```

The indexing features of MATLAB are useful for replicating a vector or a matrix:

```
>> A = [1 5]

A =

     1     5

>> A([1 2 1])

ans =

     1     5     1

>> V = [1:5]';
>> V(:,ones(3,1))   % This is known as the "Tony's trick"!

ans =

     1     1     1
     2     2     2
     3     3     3
     4     4     4
     5     5     5
```

Suppose we want to make a 5×5 matrix with all entries equal to 10. Try the following two ways:

```
>> A = 10*ones(5);          % first method
>> a = 10; A = a(ones(5));  % second method
```

Another example:

```
>> a = magic(3)

a =

     8     1     6
     3     5     7
     4     9     2
```

```
>> N = 2;
>> a(:, N(ones(1, 3)))

ans =

     1     1     1
     5     5     5
     9     9     9
```

We can use a similar approach for eliminating some of the matrix entries. For example in order to remove the NaN and Inf entries of a matrix, try the following three equivalent methods:

```
>> % first method:
>> i = find(isnan(x) | isinf(x));
>> x(i) = [];
>> % second method:
>> x(isnan(x) | isinf(x)) = [];
>> % third method:
>> x = x(~isnan(x) & ~isinf(x));
```

Check the helps for the following MATLAB functions: `meshgrid`, `repmat`, `reshape`, `find`, `any`, and `all`, for relqted items.
By combining the array indexing features with logical data-types, many of the matrix manipulations may be done without the need of for-loops and if-conditions.

5.2 Vectorized computations

We now use some of the noted MATLAB features in a few computational examples. We will make use of the following MATLAB functions: `zeros`, `ones`, `toeplitz`, `pascal`, and `hankel`.

Example 1. Signal generation

```
>> N = 5000;
>> fs = 500; % Hz
>> f0 = 3.8; % Hz
>> t = [0 : N-1]/fs;
>> x = sin(2*pi*f0*t) .* exp(-t.^2/2);
```

Example 2. Inner product

```
>> a = [1 2 3];
>> b = [4 5 6];
>> a*b'        % first method

ans =

  32

>> sum(a.*b)   % second method

ans =

  32
```

Scalars may be added, subtracted or multiplied by vectors or matrices using their corresponding operators. Vectors and matrices may also be divided by scalars in the same manner; but in order to divide a scalar by a vector or matrix, one should use the `./` operator instead of the `/` operator.
When both of the operands are nonscalar (vectors or matrices), `*`, `^`, and `/` are used for matrix operations and `.*`, `.^`, and `./` are used for operations on the entries.

Example 3. Clipping a signal

```
>> x = max(x, SOME_LOWER_BOUND); % clip from the bottom
>> x = min(x, SOME_UPPER_BOUND); % clip from the top
```

Example 4. subtraction of a vector from all the columns of a matrix

```
>> a = [1 : 5]';
>> b = eye(5);
>> b - a(:,ones(5,1))

ans =

     0    -1    -1    -1    -1
    -2    -1    -2    -2    -2
    -3    -3    -2    -3    -3
    -4    -4    -4    -3    -4
    -5    -5    -5    -5    -4
```

Example 5. Multiplying vectors in matrices Try the following four methods

```
>> N = 3000;
>> x = randn(N,1);
>> F = randn(N);
>> Y1 = x(:,ones(N,1)).*F;     % 1st in speed
>> Y2 = diag(sparse(x)) * F;   % 2nd in speed
>> Y3 = sparse(diag(x)) * F;   % 3rd in speed
>> Y4 = diag(x) * F;           % 4th in speed
```

In the above, `sparse` is a special data-type in MATLAB, which is efficient for manipulating matrices with many zero entries. Check the help for the `sparse` and `full` functions.

Example 6. Normalization of the columns of a matrix

```
>> vmag = sqrt(sum(v.^2));
>> v = v./vmag(ones(1,size(v,1)),:);
```

Example 7. First order difference

```
>> d = sin(2*pi*[0:999]*15/1000);
>> df = d(1 : end-1) - d(2 : end);
```

Example 8. Using the built-in filter function for vectorizing recursive calculations

```
>> L = 1000;
>> A = 1;
>> % bad coding style:
>> for i = 1 : L-1
>>    A(i+1) = 2*A(i) + 1;
>> end
>> % good coding style:
>>       y(i) = (tmp(N/2) + tmp(N/2+1))/2;
>> A = filter(1,[1 -2],ones(1,L));
```

Example 9. Zero-order holding of uniformly sampled data

```
>> N = 4;
>> x = [1 5 3];
>> x = upsample(x,N); % x = [1 0 0 0 5 0 0 0 3 0 0 0];
>> x = filter(ones(N,1),1,x);
x =

     1   1   1   1   5   5   5   5   3   3   3   3
```

Example 10. Zero-order holding of non-uniformly sampled data

```
>> a = 1; b = 5; c = 3;
>> x = [a 0 0 0 b 0 0 c 0 0 0 0]; % => y = [a a a a b b b c c c c c];
>> validin = find(x);
>> x(validin(2 : end)) = diff(x(validin));
>> x = cumsum(x)
 x =

     1   1   1   1   5   5   5   3   3   3   3   3
```

Example 11. Median filter of order N

```
>> x = randn(1000,1);
>> N = 20;
>> y = x;
>> if(mod(N,2)==1) % Odd length median filter
>>    for i = (N+1)/2 : length(x)-(N-1)/2
>>       tmp = sort(x(i-(N-1)/2 : i+(N-1)/2));
>>       y(i) = tmp((N+1)/2);
>>    end
>> else % Even length median filter
>>    for i = N/2 : length(x)-N/2
>>       tmp = sort(x(i-N/2+1 : i+N/2));
>>    end
>> end
```

Example 12. Moving average filter of order N

```
>> % first method
>> x = randn(1000,1);
>> N = 20;
>> y = filter(ones(N,1)/N,1,x);
>> % second method
>> y = cumsum(x)/N;
>> y(N+1 : end) = y(N+1 : end) - y(1 : end-N);
```

The second method in this example has used the following property of moving average filters:

$$H(z) = 1 + z^{-1} + z^{-2} + \ldots + z^{-N+1} = \frac{1 - z^{-N}}{1 - z^{-1}} \tag{1}$$

The last fraction in this equation is known as the *cascaded integrator comb (CIC)* filter, which is a very practical filter for DSP and FPGA implementation of down-converters and up-converters in telecommunications systems. Its efficiency is due to the fact that it only needs summations and subtractions for its implementation, which are more economic than multipliers, and a memory for keeping the last N input samples.
Although the frequency response of (**??**) is a Sinc(·) function with N lobes and an attenuation of about 13 dB in its first side-lobe (which is rather poor for a lowpass filter), by cascading several stages of such filters, better performance is achieved:

$$H(z) = \left(\frac{1 - z^{-N}}{1 - z^{-1}}\right)^R \tag{2}$$

Such filters are usually followed by a down-sample of order N. A similar filter is also used for up-conversion of signals in transmitters.

Example 13. Calculation of two-dimensional functions
Consider the two-dimensional function $F(x, y) = xe^{-x^2-y^2}$. It can be calculated efficiently as follows:

```
>> x = (-2 : 2); % arbitrary initial values
>> y = (-1.5 : .5 : 1.5); % arbitrary initial values
>> % first method. Not efficient!
>> F = zeros(length(x),length(y));
```

```
>> for i = 1:length(x),
>>    for j = 1:length(y),
>>       F(i,j) = x(i)*exp(-x(i)^2-y(j)^2);
>>    end
>> end
>> % second method. Better!
>> [X Y] = meshgrid(x,y);
>> F = X.*exp(-X.^2-Y.^2);
```

In this example we can also use the fact that F is a separable function of x and y, to further simplify the calculations. Using $F(x,y) = \left(xe^{-x^2}\right)e^{-y^2}$ we can write

```
>> F = (x .* exp(-x.^2))' * exp(-y.^2); % Efficient!
```

5.3 Profiling the program speed

We can use the `tic` and `toc` functions or the `cputime` function to find the exact execution time of a codes:

```
>> tic % start timer
>>    procedure1; % the procedure
>> toc % stop timer
>> % Or
>> t = cputime;
>>    procedure2; % the procedure
>> dt = cputime - t
```

For more detailed information on a code (including all the functions and its sub-functions), the `profile` function can be used, which gives a complete report of the program timing. This information can help one find the bottlenecks of the code and write them more efficiently or even implement them using MEX-functions, which are explained in the next section. All we need to do for profiling a program is:

```
>> profile on
>>    procedure; % a function or m-file which you want to profile
>> profile off
>> profile viewer
```

6 Linking MATLAB with external components

We can run OS commands directly from the MATLAB command line:

```
>> ! dir
>> ! autoexec.bat
```

6.0.1 Calling MATLAB routines from C or FORTRAN

MATLAB functions can be executed from a C or FORTRAN program. This is done by the MATLAB Engine library, which may be called from other programs. The corresponding examples can be found in the `<MATLAB>\extern\` directory. In summary one needs to transfer data between MATLAB and C (or FORTRAN) and to be able to execute the MATLAB functions. The MATLAB Engine has several different routines for data transfer and function calls from C (or FORTRAN).
MATLAB functions may also be converted to stand-alone programs or dynamically linked libraries (DLLs). One can use the MATLAB COM Builder (`comtool`) for this purpose. Take a look at MATLAB's documents for further details. You can also design a graphical interface for your codes using the `guide` tool.

6.0.2 Calling C routines from MATLAB

By now we have presented some general guidelines for optimizing MATLAB codes. Now suppose that we have already performed all the possible optimizations, but the program is still slow, which is common in time-consuming simulations or in real-time applications. A solution that can speed-up a program up to an order of ten times, is to rewrite the bottlenecks of the codes in C. This can be done in MATLAB by using MEX-files, which stand for MATLAB Executables.
There is of course another reason for using MEX-files. We might already have many efficient codes written in C or FORTRAN that we do not want to rewrite in MATLAB. All we have to do is to make a MEX-file interface of the source codes and call them from MATLAB.
Of course, we should note that MEX-files should not be overused, because MATLAB is a high-level programming environment for rapid system design and prototyping and one should ideally avoid going into low-level implementation details.
MEX-files are in fact DLLs made of C/C++ or FORTRAN codes, which can be executed by MATLAB. Every C MEX-file consists of the four following elements:

1. `#include "mex.h"`
2. `mexFunction`
3. `mxArray`
4. API functions

The `mexFunction` is the gateway to the DLL, which is called by MATLAB. In C, `mexFunction` always has the following form:

```
void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[]);
```

where `nlhs` and `nrhs` are the number of outputs and inputs to the function, respectively. These two integers are equivalent with the `nargout` and `nargin` built-ins in MATLAB functions. `plhs` and `prhs` are arrays of pointers to `mxArray`.
`mxArray` is a structure representing MATLAB arrays in C. All data-types are an `mxArray` structure, containing the MATLAB variable name, its dimensions, its data-type, and whether it is a real variable or a complex one. The real and imaginary parts may be accessed by the `.pr` and `.pi` fields.
There are several functions in the mex library for sending, receiving, and processing `mxArray` data. MEX-files also have the ability of loading variables directly from the caller function or the base workspace of MATLAB. Note that MATLAB functions may also be directly called from MEX-files:

```
mexCallMATLAB(nlhs,plhs,nrhs,prhs,"MATLAB Function Name");
```

This will however reduce the speed efficiency of MEX-files due to the overhead of calling MATLAB routines and the required data transfer.
The stages of generating MEX-file DLLs from an existing MEX-file is as follows:

1. C compiler selection by running `mex -setup`. MEX-files may be compiled using any C compiler.
2. MEX-file DLL generation, by calling the mex command as follows:

   ```
   >> mex example.c
   >> % or
   >> mex example.c objfile.obj ex1.c libfile.lib
   ```

   As you see, the `mex` command can also take multiple C-files and pre-compiled objects and libraries.
3. calling the MEX-file from MATLAB:

   ```
   >> example([1 2],1:10);
   ```

If you are using Visual Studio or similar packages, you can use the `dumpbin.exe` program to check the contents of the DLLs produced by MATLAB.

References

[1] *Documentation for MathWorks Products R2020a*. [Online]. Available: https://www.mathworks.com/help/, Aug 2020.

[2] D. Eyre, *MATLAB Basics and a Little Beyond*. [Online]. Available: https://www.math.utah.edu/~eyre/computing/matlab-intro, 1998.

[3] Etter, Delores Maria, David C. Kuncicky, and Douglas W. Hull, *Introduction to MATLAB*, Prentice Hall, 2002.

[4] Pascal Getreuer (2020). *Writing Fast MATLAB Code*, https://www.mathworks.com/matlabcentral/fileexchange/5685-writing-fast-matlab-code, MATLAB Central File Exchange. Retrieved August 6, 2020.

[5] P.J. Acklam, *MATLAB array manipulation tips and tricks*. [Online]. Available: https://www.coursehero.com/file/14588631/MATLAB-array-manipulation-tips-and-tricks/, October 2003.

[6] J.R. Gilbert, C. Moler, and R. Schreiber, *Sparse Matrices in MATLAB: Design and Implementation*. [Online]. Available: https://doi.org/10.1137/0613024, October 1991.

[7] *Techniques to Improve Performance*. The Mathworks support center [Online]. Available: http://www.mathworks.com/support/tech-notes/1100/1109.html.

[8] B. Shah, *A Tutorial to Call MATLAB Functions from Within A C/C++ Program*. [Online]. Available: https://www.scribd.com/document/76938035/A-Tutorial-to-Call-MATLAB-Functions-From-Within-a-C-C-Program.

[9] *External Language Interfaces*. The Mathworks support center [Online]. Available: http://www.mathworks.com/support/tech-notes/1600/1605.html.