

Matlab DOs and DON'Ts

Dmitri Nikonov

“the unrepentant Matlab programmer”

Dmitri.e.Nikonov@intel.com

Outline

Matlab positioning among simulation tools

Tools: profiler, interrupt debug, windiff

Tips: loops vs. vectorization

Tips: vector function calls

Tips: sparse matrices

Tips: self-made vs. built-in

Program structuring: modularization, encapsulation, memory management

My philosophy on Matlab

Matlab positioning

Perceived as a tool for small calculations (“homework in a graduate course”)

In fact can handle most complex simulation projects

On par with C – structures and classes, function overloading, inheritance, GUI capabilities and many more – not the topic of this lecture

Benefits compared to C – natural use of complex numbers, unique operations with matrices, powerful library of functions, efficient built-in numerical algorithms (need to be left to professionals), integrated plotting

Can be the tool of choice for scientific simulations

Matlab suffers from improper use (esp. C or Fortran habits) – the topic of this lecture

Tools: profiler

Parent functions:	
sparsesolu	1
Child functions:	
none	

100% of the total time in this function was spent on the following lines:

```

          78:    % Create a sparse matrix from its diagonals
0.00000087 0% 79:    B = arg1;
          80:    d = arg2(:);
          81:    p = length(d);
0.00000006 0% 82:    moda = (nargin == 3);
          83:    if moda

          90:    % Process A in compact form
0.01000000 11% 91:    [i,j,a] = find(A);
0.0003260 0% 92:    a = [i j a];
0.01000000 11% 93:    [m,n] = size(A);
          94:    for k = 1:p
```

Times the run
Enables you to find
the bottlenecks in the
program
See which function
called which and how
many times
Invoked as follows

```
profile on
chloop = 'slow';
simatr = 1000;
a = rand(simatr,simatr);
b = rand(simatr,simatr);
c = suexpo(a,simatr,chloop);
profile report
```

Tools: interrupt debug

Invoked by placing a line

keyboard

at some point in the Matlab program

Allows you to examine the workspace of the function normally not available,

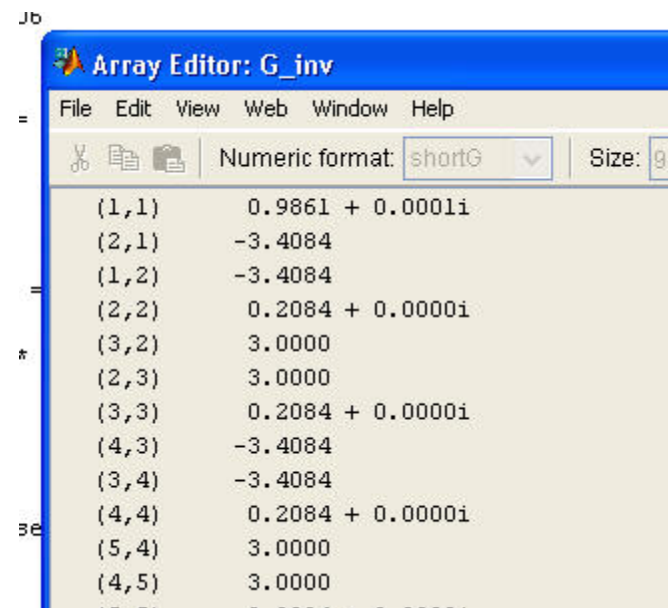
Execute any Matlab commands to find what goes wrong

K>> return

To go back to execution

K>> dbquit

To interrupt execution



The screenshot shows the MATLAB Array Editor window for a variable named 'G_inv'. The window title is 'Array Editor: G_inv'. The menu bar includes 'File', 'Edit', 'View', 'Web', 'Window', and 'Help'. The numeric format is set to 'shortG' and the size is 94. The array is displayed as a 5x5 matrix with the following values:

(1,1)	0.9861 + 0.0001i
(2,1)	-3.4084
(1,2)	-3.4084
(2,2)	0.2084 + 0.0000i
(3,2)	3.0000
(2,3)	3.0000
(3,3)	0.2084 + 0.0000i
(4,3)	-3.4084
(3,4)	-3.4084
(4,4)	0.2084 + 0.0000i
(5,4)	3.0000
(4,5)	3.0000

Tools: windiff

Part of Microsoft Visual Studio
Extremely useful to track changes, look for bugs

```

c:\denikono\purdue\moscnt.1 : c:\denikono\purdue\moscnt.4
1  \.anti_dummy.m          identical
2  \.benchmark.m          only in c:\denikono\purdue\moscnt.4
3  \.benchmark_results.mat only in c:\denikono\purdue\moscnt.4
4  \.charge.m            identical
5  \.current.m           identical
6  \.define_f_prime.m    only in c:\denikono\purdue\moscnt.4
7  \.define_inputs.m     different (c:\denikono\purdue\moscnt.4 is more recent)
8  \.define_inputs_benchmark.m only in c:\denikono\purdue\moscnt.4
9  \.dufu1.m             only in c:\denikono\purdue\moscnt.4
10 \.dufu2.m             only in c:\denikono\purdue\moscnt.4
11 \.dummy.m             different (c:\denikono\purdue\moscnt.4 is more recent)
12 \.func_charge.m       different (c:\denikono\purdue\moscnt.4 is more recent)
13 \.func_current.m     identical
14 \.integral.m         identical
15 \.mos_results-20-oct-2004.mat only in c:\denikono\purdue\moscnt.1
16 \.mos_results.mat    only in c:\denikono\purdue\moscnt.1
17 \.moscnt_main.asv    only in c:\denikono\purdue\moscnt.1
18 \.moscnt_main.m      different (
19 \.myquad.m           identical
20 \.plot_results.m     only in c:\denikono\purdue\moscnt.4
21 \.poisson.m         different (

97      spB_d=sparse(Np,1); spB_d(Np)=1;
98      Hdiag=Emcnt;
99      Hdiag(1)=Hdiag(1)+con_s;
100     Hdiag(Np)=Hdiag(Np)+con_d;
100 <|   G_inv=sparse(diag(ep-Hdiag))+sparse(diag(AUD,1))+sparse(diag(AUD,1)');
      !>   eH = ep - Hdiag;
      !>   AUD_L = [0; AUD];
      !>   AUD_R = [AUD; 0];
      !>   eHAUD = [AUD_R eH AUD_L];
      !>   G_inv = speye(Np,Np);
      !>   G_inv = spdiags(eHAUD,-1:1,G_inv);
101     G_s=G_inv\spB_s;
102     G_d=G_inv\spB_d;
103     LDOS_S=-abs(G_s).^2*imag(con_s)*2;
104     LDOS_D=-abs(G_d).^2*imag(con_d)*2;
      !>   %keyboard
105     end
106
107     Nspec=0.5*(sign(ee-Emcnt)+1).*(f_2*LDOS_D+f_1*LDOS_S)...
108 <|   -0.5*(sign(Emcnt-ee)+1).*((1-f_2)*LDOS_D+(1-f_1)*LDOS_S);
[108] !>   -0.5*(sign(Emcnt-ee)+1).*((1-f_2)*LDOS_D+(1-f_1)*LDOS_S);
109 <|
110 <|
111 <|

```



Loops

```
function c = multi(a,b,simatr,chloop)
switch chloop
  case 'slow'
    for j = 1:simatr
      for k = 1:simatr
        c(j,k) = a(j,k)*b(j,k);
      end
    end
  case 'fast'
    c = a.*b;
end
```

SLOW 139s

FAST 0.1s

You shall not use a “for”-loop when you can use a vectorized operation

Function call

```
function c = suexpo(a,simatr,chloop)
switch chloop
  case 'slow'
    for j = 1:simatr
      for k = 1:simatr
        c(j,k) = exp(a(j,k));
      end
    end
  case 'fast'
    c = exp(a);
end
```

SLOW 135s

FAST 0.4s

You shall not call a function element-by-element when you can call a function of a matrix

Sparse matrices

```
function sparsesolu(simatr,chloop)
di1 = rand(simatr,1);
di2 = rand(simatr,1);
di3 = rand(simatr,1);
b = rand(simatr,1);
switch chloop
  case 'slow'
    a = zeros(simatr,simatr);
    di1red = di1(1:end-1);
    di3red = di3(2:end);
    a = sparse(diag(di2)) + sparse(diag(di1red,-1)) + sparse(diag(di3red,1));
    c = a\b;
  case 'fast'
    a = speye(simatr,simatr);
    % starting with an empty matrix does not work for some reason
    ditot = [di1 di2 di3];
    a = spdiags(ditot,-1:1,a);
    c = a\b;
end
```

SLOW 12s

FAST 0.1s

You shall not form a full matrix and then try to convert to a sparse one

Recursive algorithms

Function 'tridiag' copied with adaptation from "Numerical Recipes" book

```
function u = tridag(a,b,c,r,n)
bet=b(1);
u(1)=r(1)/bet;
for j=2:n      % decomposition and forward substitution
    gam(j)=c(j-1)/bet;
    bet=b(j)-a(j)*gam(j);
    u(j)=(r(j)-a(j)*u(j-1))/bet;
end
for j=n-1:-1:1 % backsubstitution
    u(j)=u(j)-gam(j+1)*u(j+1);
end
```

SLOW 29s

FAST 0.5s

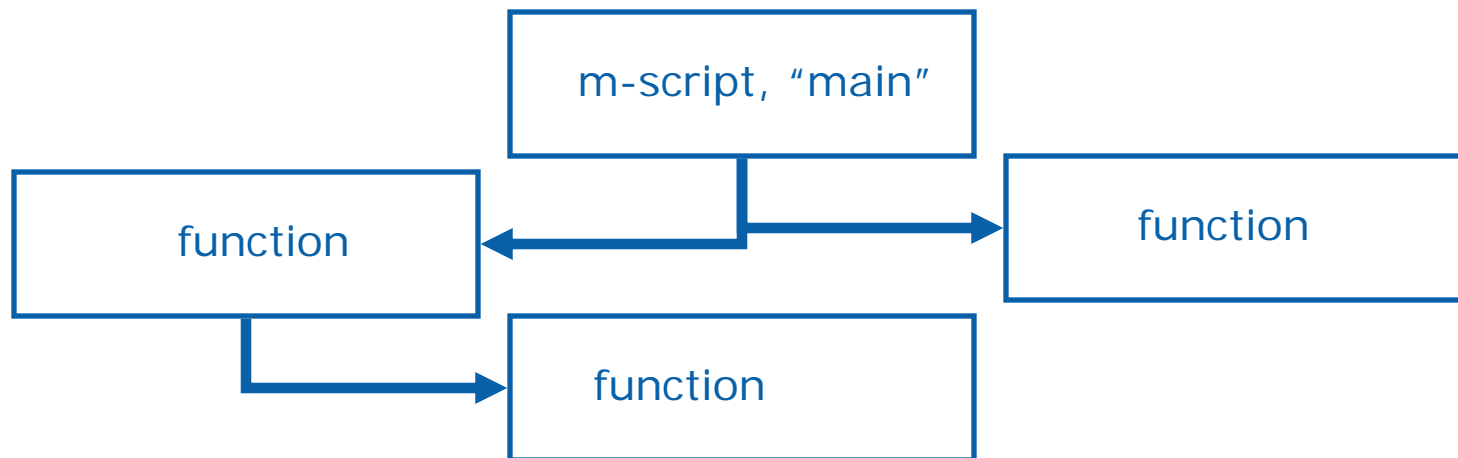
You shall not use "clever" algorithms which cannot be vectorized

Program structure: modularization

Make any piece of code which performs a separate task into a function. It will be easy to improve and re-use it

Do not use M-scripts except for the “main” module – scripts do not let you control input and output variables

In any case, do not use a module longer than 100 lines – it is impossible to improve later



You shall break your program in many small functions

Program structure: documentation

Write many more comments than you consider sufficient.

>> help block2full will show the set of lines just after the 'function'.

```
function Fmat = block2full(Bdmat,Np,Nc,ndiag)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% function Fmat = block2full(Bdmat,Np,Nc,ndiag)
% convert a block-diagonal form of matrix to full one
% Bdmat = matrix in the band-diagonal form
% Fmat = matrix in the full form
% Np = number of blocks/slices
% Nc = number of elements in a block/slice
% ndiag = index of a block-diagonal,
% =0 -- main, <0 -- lower, >0 upper diagonals
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Lmat = Np-abs(ndiag);           % number of blocks on the diagonal
Fmat = zeros(Np*Nc,Np*Nc);     % size of the full matrix
for k=1:Lmat
    jna = 1 + (k-1)*Nc;        % beginning horizontal index for a block
    if(ndiag<0)
        jna = jna - ndiag*Nc;  % correct if lower diagonals
    end
    jko = jna + Nc - 1;        % ending horizontal index
    ina = jna + ndiag*Nc;      % beginning vertical index
    iko = jko + ndiag*Nc;      % ending vertical index
    Fmat(jna:jko,ina:iko) = Bdmat(:,k); % fill in the block
end
```

Really diligent
programmers insert
authors' name, version
history, copyright

You shall comment every line of your code



Program structure: variable scope

The benefit of the function – tight control of input and output arguments. You avoid a lot of errors with unintended modification of data in other pieces of your code.

Encapsulation: No variable with the same name defined in other parts of your (or somebody else's) code can be confused with local variables. They are only defined in the scope of this function.

'global' variables defeat this useful feature. Do not use to pass input and output arguments.

Avoid them in most cases !!! Exceptions:

a) global constants. You do not want to assign them in many parts of your code and chase all of them to change a value

b) a library function that calls a function

```
[t,yevol] = ode113('urav',[tsta tfin],y,opa);
```

and you need to change the names and number of arguments passed to the lower level function ('urav' in this case)

You shall not use global variables with few necessary exceptions

Memory management

Three type of memory a program may use

- on chip cache
- RAM
- hard drive

In the order of decreasing speed but increasing capacity RAM storage is fast enough. If your variable space does not fit in RAM, then the program will spool on hard drive. It makes execution **EXTREMELY SLOW.**

Get as much RAM for your computer as feasible.

Check the variable space with

>> whos

If the matrices do not fit in RAM, re-do your algorithm, free memory with a command in the code, e.g.

clear MyBigMat

or make your simulation less ambitious (less variables).

You shall not use array size that does not fit in you RAM

My philosophy on Matlab

I admit that optimized C will be faster than optimized Matlab

But Matlab will save you on development, maintenance, and re-use time

If you write Matlab in the style of C, performance will be horrible

Need to use unique Matlab features to optimize the program

Do not forget general rules of good programming: structuring your program, control scope of variables, manage memory

