Notes on efficient Matlab programming

Dag Lindbo
dag@csc.kth.se

January 27, 2009

1 Introduction - is Matlab slow?

It is a common view that Matlab is slow. This is a very crude statement that holds both some merit and some error. The truth is that the core Matlab language is designed for a particular type of task (hence the name Matrix Laboratory). Thus, it shall not be confused with a general purpose programming language like C++. Most of those who hold the view that Matlab is slow simply write bad Matlab code. Unlike in Fortran, it is very easy in Matlab to write a program that runs really slowly. However, if one uses Matlab for what it is meant to do then one only needs to use a few basic techniques to get efficient code. This text will outline these by way of example.

A valid question is whether Matlab code efficiency matters. It is certainly possible to get 10, 100, or even 1000 times speedup simply by avoiding some common efficiency pitfalls. Often, a well written Matlab program is easier to read and less prone to bugs. So there is really very little to lose by learning how to use the core Matlab language efficiently.

The conclusions in this text are based on performance results obtained with Matlab R2007b, running single thread on a Intel E6600 (Conroe) CPU with a 32-bit Linux OS. Efficiency results are likely to be platform, and Matlab version, dependent.

Conclusion 1: Poor Matlab code will run slowly. Bad programmers write code that runs slowly and blame Matlab instead of themselves.

Conclusion 2: Matlab is computationally efficient for matrix and vector operations. Recent additions, namely the JIT compiler, may accelerate code with loops and element-wise logic and manipulation. Matlab should still not be regarded as a general purpose language, like C++.

2 Tools of the trade

2.1 The Profiler

The Matlab profiler is an awesome tool for measuring the performance of a program. In profiling mode, Matlab counts how many times each individual statement

in a program is executed and reports the amount of time spent on each line. This identifies bottlenecks in run time.

There are two ways to use the profiler:

a) Open the profile viewer by calling `profview` in the command window. In the field "run this code" type the name of your program and hit enter. This program is run in the profiler and once it finishes the profiling data is displayed.

b) In the command window call `profile on` to start profiling. Then run your program. After it is done, do `profile report`. This method is more cumbersome.

Some examples of profile reports are given for the examples later in this document.

2.2 MLint

`lint` is an old tool used on Sun systems to inspect source code (mainly C). It has passed a bit from prominence, but a derivative of it exists in Matlab. `mlint` is a tool that inspects m-files and warns of potential problems. These warnings and suggestions include that

- a value assigned to a variable is not used
- a global variable is declared but not used
- the code uses & instead of && for scalar `AND` conditionals
- the code compares the length of an array to 0, instead of calling the `isempty` function.
- hundreds more...

Most MLint warnings are not about code efficiency but many are! It is a good habit to use this tool to find these problems, which are otherwise easy to overlook. Note that MLint output is included in profile reports.

3 Efficient techniques

This is the main part of this document. Writing a good Matlab is not as much about optimization as it is about avoiding common mistakes. Here we shall illustrate some of the most important efficiency techniques, including

- Preallocation of arrays
- Anonymous and in-line functions
- Vectorization via element sub-scripting
- Vectorization via logical index masking

We will also explain the concept of copy-on-write and discuss construction of sparse matrices. These will all be covered in a few examples.

3.1 Example: Euler's explicit method for ODE systems

Efficiency topics: Preallocation and anonymous functions.

We have a system of ordinary differential equations,

$$\frac{du}{dt} = f(t, u), \quad u(0) = u_0.$$

Consider the task of implementing the simplest of all schemes for computing an approximate solution to a system of ODEs: the Euler Forward method.

$$u_{n+1} = u_n + hf(t_n, u_n), \quad u_0 \text{ known}$$

Listing 1: First implementation of Euler

```
2  clear all, close all
3
4  % Paramters
5  func = inline('[x(1)+1 x(2)-1]','t','x');
6  t_0 = 0;
7  t_end = 3.02157;
8  h = 0.0001;
9  u_0 = [0  0];
10
11 Nt = round((t_end-t_0)/h)+1; % make sure we hit t_end
12 h = (t_end-t_0)/(Nt-1);
13
14 % IC
15 t = t_0;
16 u = u_0;
17
18 % Time-stepping
19 T = t;
20 U = u;
21 for i = 2:Nt
22
23   % Explicit Euler
24   u = u + h*func(t,u);
25   t = t+h;
26
27   % Accumulate data
28   T = [T ; t];
29   U = [U ; u];
30 end
```

What is wrong with this program? Take a look at a profile report given in Appendix 1. Which lines are most costly?

Line 22: Evaluation of the function.

The problem here is that the program defines an inline function, `func`, which is evaluated many times. This facility is inefficient, and should be replaced by an anonymous function instead.

Line 26-27: Accumulation of results

This is one of the capital sins of Matlab programming. In each iteration the array grows, and that requires that Matlab allocate memory for a larger array which is *very* costly. The solution is to pre-allocate an array that will be filled in the loop.

Listing 2: Improved implementation of Euler

```
2  clear all, close all
3
4  % Paramters
5  func = @(t,x) [x(1)+1 x(2)-1];
6  t_0 = 0;
7  t_end = 3.02157;
8  h = 0.0001;
9  u_0 = [0 0];
10
11 Nt = round((t_end-t_0)/h)+1;
12 h = (t_end-t_0)/(Nt-1);
13
14 % IC
15 t = t_0;
16 u = u_0;
17
18 % Pre-allocate storage
19 T = zeros(Nt,1);
20 U = zeros(Nt,length(u));
21
22 % Time-stepping
23 T(1) = t;
24 U(1,:) = u_0;
25 for i = 2:Nt
26
27   % Explicit Euler
28   u = u + h*func(t,u);
29   t = t + h;
30
31   % Accumulate data
32   T(i)   = t;
33   U(i,:) = u;
34 end
```

Take a look at the profile report for `euler_ode_fast.m`. The total run time dropped from 16.3 to 0.6 seconds. This is a speedup of over 25 times!

Conclusion 3 Always pre-allocate storage for arrays.

Conclusion 4 Prefer anonymous functions over the `inline` construction.

3.2 Example: Evaluating a piecewise function

Efficiency topics: Vectorization, array subscript and logical referencing.

Consider the task of evaluating a piecewise defined function,

$$f(t) = \begin{cases} 1, & t < 1 \\ 2, & 1 \leq t < 2 \\ 0, & t \geq 2 \end{cases}$$

for a vector of t-values. We recall the lesson from the previous section about preallocation of storage and write the following program:

Listing 3: First implementation of piecewise function evaluation

```
1  f = zeros(N,1);
2  for i = 1:N
3    if t(i) < 1
4      f(i) = 1;
5    elseif t(i) >= 1 && t(i) < 2
6      f(i) = 2;
7    end
8  end
```

For a programmer with experience from C this would be a very natural way of implementing the function. In Matlab, however, it is shameful! We need to express the operation in terms of vectors, not individual elements. There is a handy function called `find` which returns a list of indices for which the logical statement in its argument is true. Example: `find( (5:10) > 7)` returns a vector `[4 5 6]`. This leads to a second attempt at the evaluation:

Listing 4: Second implementation of piecewise function evaluation

```
1  f = zeros(N,1);
2  f( find(t < 1) ) = 1;
3  f( find(t >= 1 & t < 2) ) = 2;
```

This implementation is pretty good Matlab code. The `find`-call returns an array of indices that are assigned to. We may refer to this as subscript referencing. It is good, but we can do better. Recently, the concept logical referencing was introduced. The statement `(5:10) > 7` by itself returns a logical array `[0 0 0 1 1 1]`. That is, the logical statement $x > 7$ is false for the first three elements in the vector `[5 6 7 8 9 10]`, but true for the last three. The logical array can be used as a mask, leading to the third attempt at the piecewise evaluation:

Listing 5: Third implementation of piecewise function evaluation

```
1  f = zeros(N,1);
2  f( t < 1 ) = 1;
3  f( t >= 1 & t < 2) = 2;
```

There is a final alternative which is shorter. We can convert the logical array to a float array:

Listing 6: Fourth implementation of piecewise function evaluation

```
1  f = ( t < 1 ) + 2*( t >= 1 & t < 2 );
```

Implementation	Run-time (sec)
Loop	28
Find	0.8
Logical referencing	0.4
Logical conversion	0.4

Table 1: Run time comparison between four implementation of the evaluation of a piecewise defined function. $N = 2000$, repeated 10000 times

Now we can be really proud of our Matlab skills! But which one is faster? To test this we let $N = 2000$ and run each evaluation 10000 times. The results are in Table 1. The speedup from vectorization is roughly 70 times! Note that in this example, we have vectorized two tasks at the same time: the assignment and the conditional.

Conclusion 5: It is really important to vectorize computations!

Conclusion 6: Logical indexing is faster than `find`.

It is highly recommended to try to vectorize some other computations. Suggestions:

- Evaluate a piecewise function which is not constant on the intervals. All four implementations are possible.
- Implement Gaussian elimination without pivoting. Can you write it with only one loop?

3.3 Example: Evaluating a symmetric difference formula

Efficiency topics: Direct and indirect referencing, just-in-time acceleration.

The performance results seen in the previous section clearly reveal the preference Matlab has for vectorization. This used to be a black-and-white issue: loops are bad, vectorization is good. Since version 6.5, and the introduction of a just-in-time (JIT) compiler, the situation is more complex.

To see this take a simple vectorization task: applying a symmetric difference formula, commonly called D_0, to a grid function in a periodic domain,

$$(D_0 f)_i = \frac{1}{2h}(f_{i+1} - f_{i-1}), \quad i = 1, 2, \ldots, N$$
$$f_{i+N} = f_i.$$

The periodic boundary condition means that the edges of the domain must be treated separately. This lures programmers into writing loopy code that treats all elements one by one. Something like this typical:

Implementation	Run-time (sec)
Loop	2.7 (267 without JIT)
First vectorized	3.1
Second vectorized	2.7
Third vectorized	5.3 (2.6 with integer cast)

Table 2: Timing results for four implementations of evaluation a symmetric difference formula. $N = 10000$, and 10000 repetitions.

Listing 7: Loop implementation of D_0

```
1   df = zeros(size(f));
2   df(1) = (f(2)-f(N))/(2*h);
3   for i = 2:N-1
4     df(i) = (f(i+1)-f(i-1))/(2*h);
5   end
6   df(N) = (f(1)-f(N-1))/(2*h);
```

Most experienced Matlab users will get justifiably twitchy and nervous about the `for`-loop (remember the previous example). A natural first step is to vectorize the loop, yielding the second implementation

Listing 8: First vectorized implementation of D_0

```
1   df = zeros(size(f));
2   df(1) = f(2)-f(N);
3   df(2:N-1) = f(3:N) - f(1:N-2);
4   df(N) = f(1)-f(N-1);
5   df = df/(2*h);
```

This can be vectorized further into more compact code, as

Listing 9: Second vectorized implementation of D_0

```
1   fp = [f(2:N); f(1)];
2   fm = [f(N); f(1:N-1)];
3   df = (fp - fm)/(2*h);
```

or by precomputing permuted index vectors

Listing 10: Third vectorized implementation of D_0

```
1   ip = [2:N 1];                 %% should cast to int, e.g.
2   im = [N 1:N-1];               %% im = int32([N 1:N-1]);
3   df = (f(ip) - f(im))/(2*h);
```

The latter alternative has a striking appeal: the index vectors, `ip` and `im` are independent of f (they depend only on the D_0 scheme). Thus, D_0 can be applied to many f (same size) without recomputing the index vectors. These compact implementations make a noteworthy sacrifice: the additional memory required for two intermediate vectors. In this case, where we are dealing with vectors and not arrays of higher dimension, this is not likely to be a major concern.

To do some performance testing, let $N = 10^5$ and repeat 10^5 times. In the last implementation, the computation of the index vectors is not repeated. The timing results are in Table 2. Here, two things are striking:

The loopy code is fast (in recent Matlab).
This is entirely due to the addition of the JIT compiler to Matlab. Note that without this feature (as in older versions), the computation takes a hundred times longer. How come the JIT did not speed up the evaluation of the piecewise function? A qualified guess is that it failed to see that the conditionals are independent of the calculations (only depending on t).

The implementation with precomputed index vectors is slow.
Here, the reason is more subtle: we are seeing the difference between direct and indirect indexing. What does this mean? The statement `a(1:6)` means "from the vector `a`, extract elements one through 6". On the other hand, `a(idx)` means "from `a` extract the elements specified in `idx`". Matlab can optimize the indexing more if the comma is in the indexing statement. It helps to manually cast the index array to an integer format first, making this the most efficient implementation (verified with a range of N and repetitions).

Conclusion 7: When precomputing index arrays, cast them to `int32`.

Conclusion 8: The JIT compiler can accelerate a loopy code to compete with good vectorization. This effect should not be relied on, so use the Profiler!

3.4 Example: Passing a matrix to a function

Efficiency topics: Pass by reference and copy-on-write

When you pass data as input to a matrix you can edit that data inside the function without changing the value outside. This is called pass-by-value. You simply get a local copy inside the function name-space. However, if the argument to a function is a large matrix, then the amount of work and memory to copy the matrix may make this approach infeasible. Fortunately, Matlab has a smart solution to this, referred to as copy-on-write. To illustrate what is going on, consider the following function:

Listing 11: Copy-on-write when passing data to a function

```
1  function pass_by_ref()
2
3  N = 8000;
4  A = ones(N,N);
5
6  sum_matrix(A);
7  change_and_sum(A);
8
9  function s = sum_matrix(mat)
10 s = sum(sum(mat));
11
12 function s = change_and_sum(mat)
13 mat(1,1) = 4000;
14 s = sum(sum(mat));
```

The two sub-functions `sum_matrix` and `change_and_sum` are identical, aside from the assignment that the latter does before the summation. In the code it looks like the first functions does $N \cdot N$ operations, and the second does $N \cdot N + 1$. However, take a look at the profile report for this program (given as an Appendix).

The call to `change_and_sum` takes four times longer. A further inspection of the profile report (not shown) reveals that the assignment statement on line 13 takes 75% of the time spent in `change_and_sum`. Why?

Since the matrix `mat` in `sum_matrix` is unchanged, it is never copied. In `change_and_sum`, the assignment into a single element of `mat` triggers the copying of the entire matrix. Hence the term copy-on-write.

Conclusion 9: If input to a function is unchanged, this data is passed to the function as a reference (no copy). An assignment triggers a copy operation, which can be costly.

3.5 Example: Constructing a tridiagonal sparse matrix

Efficiency topics: Sparse matrix data layout

It is very common in numerical treatment of differential equations that the discretization of the equation leads to a sparse matrix (e.g. FEM, FDM). In time-dependent problems, this matrix is likely to change in each time-step. A sparse matrix is a complicated data structure, and explaining how it is done in Matlab is beyond our scope here (see Further Reading). It is important to realize that one can not treat a sparse matrix the same way as a full one. To see this we take the construction of a tridiagonal matrix with -2 on the main diagonal and 1 above and below it. The first implementation may be something like

Listing 12: First implementation of creating a tridiagonal matrix

```
1   A = sparse(N,N);
2   A(1,1) = -2;
3   A(1,2) =  1;
4   for i = 2:N-1
5     A(i,i-1) =  1;
6     A(i,i)   = -2;
7     A(i,i+1) =  1;
8   end
9   A(N,N-1) =  1;
10  A(N,N)   = -2;
```

There are a few problems with this code:

Line 1 No preallocation of storage.

Lines 4-8 The loop order is by row. Since Matlab stores data by column it is faster to loop over columns.

The second implementation fixes these issues

Listing 13: Second implementation of creating a tridiagonal matrix

```
A = sparse([],[],[],N,N,3*N);
A(1,1) = -2;
A(2,1) =  1;
for i = 2:N-1
  A(i-1,i) =  1;
  A(i,i)   = -2;
  A(i+1,i) =  1;
end
A(N-1,N) =  1;
A(N,N)   = -2;
```

This is still bad Matlab code! As it turns out these changes do not lead to any speed-up of the code, so there is something else going on: We are not mindful of what Matlab has to do to insert an element into a sparse matrix, namely a lot of searching and sorting. The preferred approach is totally different. We create three vectors, call them `J`, `K` and `X` such that `A(J(i),K(i))=X(i)`. Then a sparse matrix can be created from these vectors. This is done in the third implementation:

Listing 14: Third implementation of creating a tridiagonal matrix

```
J = zeros(3*N-2,1);
K = J;
X = J;
J(1) = 1; K(1) = 1; X(1) = -2;                  %A(1   ,1)=-2
J(2) = 2; K(2) = 1; X(2) =  1;                  %A(2   ,1)= 1
k = 3;
for i = 2:N-1
  J(k)   = i-1; K(k)   = i; X(k)   =  1;        %A(i-1,i)= 1
  J(k+1) = i;   K(k+1) = i; X(k+1) = -2;        %A(i   ,i)=-2
  J(k+2) = i+1; K(k+2) = i; X(k+2) =  1;        %A(i+1,i)= 1
  k = k+3;
end
J(end-1) = N-1; K(end-1) = N; X(end-1) =  1; %A(N-1,N)= 1
J(end  ) = N;   K(end  ) = N; X(end  ) = -2; %A(N-1,N)=-2

A = sparse(J,K,X);
```

The final thing we can do is to use the built-in function `spdiags`,

Listing 15: Fourth "implementation" of creating a tridiagonal matrix

```
e = ones(N,1);
A = spdiags([e -2*e e],[-1 0 1],N,N);
```

To get a feel for the relative efficiency of these four implementations, take a large enough N, say $N = 2 \cdot 10^5$ (which is still pretty small in e.g. FEM). The timing results are in Table 3. We see a speed-up of a factor ~ 700 for modest size! For bigger matrix, $N = 5 \cdot 10^6$ it was not even possible to obtain timing results for the first two implementations, see Table 4. It is perhaps a bit surprising that we managed to beat the built-in function by a factor ~ 3. In all fairness though, `spdiags` can easily generate any number of diagonals. It should also be noted that it may not be worth the effort to try to beat the built-in function.

Conclusion 10: Any insertion of an element into a sparse matrix is really costly!

Implementation	Run-time (sec)
Loop by row	58.4
Loop by col	56.3
Index vectors	0.08
`spdiags`	0.22

Table 3: Run-time for four different implementation of creating a sparse tridiagonal matrix. $N = 2 \cdot 10^5$.

Implementation	Run-time (sec)
Loop by row	???
Loop by col	???
Index vectors	1.9
`spdiags`	6.1

Table 4: Run-time for four different implementation of creating a sparse tridiagonal matrix. $N = 5 \cdot 10^6$.

Conclusion 11: It is possible in some cases to write quicker versions Matlab's built-in functions. This can be time-consuming though, so be careful!

4 Further Reading

- *Writing Fast Matlab Code*, Pascal Getreuer
  `www.math.ucla.edu/~getreuer/matopt.pdf`
- *Matlab Technical Documents*, The Mathworks
  `www.mathworks.com/support/tech-notes/list_all.html`
- *Sparse Matrices in Matlab: Design and Implementation*,
  Gilbert JR, Moler C and Schreiber R
  `www.mathworks.com/access/helpdesk/help/pdf_doc/otherdocs/simax.pdf`