Good Matlab Programming Practices
for the Non-Programmer

1. READABILITY & ORGANIZATION

1.1 Use structures instead of globals; pass the structures to and from functions. This keeps the functions modular and the variables obvious. It also helps organize the variables.

1.2 Use strings and `SWITCH` as a proxy for enumerated types, it's easy to use, fast and very readable:

```
fruit='apple';
switch(fruit)
   case 'apple', fun1;
   case 'banana', fun2;
end;
```

1.3 Use `EVAL, FEVAL, EXIST`, and other self modifying code sparingly. It may be really fun and very flexible, but its very confusing if you're not the one debugging it. Do not use `EXIST` to repair faults in programming logic.

1.4 Avoid MEX files when possible; K.I.S.S. (Keep It Simple, Stupid). Most of the time, vectorization is the appropriate solution and it allows us to keep most everything in one language, open and easily modified.

1.5 Use long variable names with underscores; it helps me avoid a great deal of documentation (ie. `delivery_yield_using_unwind_RP`)

1.6 Make good use of comments! Try using headers. Avoid the `if(0)` construct, it is difficult to find, especially if you are using an editor that offers syntactical highlighting (ie. Medit , Emacs, Xemacs, TextPad, Nedit).

1.7 It's nice to line up equal signs, spaces, and commas; this makes it easy to spot errors in similar statements.

1.8 Adopt a guideline for variable usage and commenting style. A common coding style is NOT necessary, as many programs (ie. Emacs) can reformat style easily.

1.9 Use all capitals for global-type variables, ie. flags and comments.

1.10 Keep functions small. They may be re-used by other functions; besides it makes them easier to read and helps keep you organized.

1.11 If a function is used only by one other function, include them in the same file.

1.12 Use "graceful degredation;" function outputs should include an error code. (please continue adding to this thread if you want).

1.13 You may want to assure the shape of a vector (provided you know it's a vector), by using it like this: `myfun(x(:))`. You can check if x is a vector by `~isempty(x) & ( sum(size(x)>1) <= 1 )`.

1.14 Separate any generic code into another small function.

1.15 Keep lines short by using the continuation character (ellipsis).

1.16 Large indents can help late at night when those 2-space indents are hard to see.

1.17 It is not important that the function name match the file name, but it makes life easier.

1.18 Stick to one capitalization scheme throughout your program. On NT you can get creative with how you call your functions since windows doesn't check case when it goes searching for files. This brings up platform independence problems because UNIX is case sensitive.

2. SPEED

2.1 Use structures of arrays, not arrays of structures, unless it is important to your code. Arrays of structures are slow. Avoid `DEAL`, it's often slow. For example, converting data of the form

```
t(1).rt_coupon=.045;
t(2).rt_coupon=.050;
t(3).rt_coupon=.060;
```

by typing `s.coupon=[t.rt_coupon]` can produce great speed advantages

2.2 There is no need to duplicate data. Instead of using `s.coupon.TU1=[5.625 5.500 5.000]` and `s.coupon.FV1=[5.500 6.000]` you can do this:

```
s.coupon=[5.625 5.500 5.000 6.000]
TU1_index=[1 2 3];
FV1_index=[2 4];
```

Now you can reference `s.coupon(TU1_index)` and `s.coupon(FV1_index)`.

2.3 Preallocate when possible, it enhances speed and often makes the code easier to read. But note, if You declare `ZEROS` or `ONES` matrices (homogeneously) MATLAB may not improve the speed but if you use other special function (such as `RAND` or `LINSPACE`) it may help.

2.3 Creative use of `FIND, PROD, SUM, CUMSUM, NaN, REPMAT, RESHAPE, ONES` and `ZEROS` can really help vectorize your code, but try not to make the code too cryptic. There is a tradeoff between execution time and readability. If your statement is cryptic, either include an equivalent (but slow and easy to read) version in a comment or write a MEX file.

2.4 Use `PROFILE` to identify where you need to work on speed improvements.

3. EASE OF USE

3.1 Rather than populate variables with assignments (`X=1, Y=3`, etc), read in a CSV file. This allows an inexperienced user to modify these values without having to touch the code.

4. GENERAL GOOD PRACTICE

4.1 Use `TRY, CATCH, END` generously. Error checking is important. It is far better to anticipate errors, but at least your code won't blow up.

4.2 Using the interpreter is too easy! Serious code review/reading, Sometimes printing the routine and reading it should precede machine execution. This is the best defense against that class of bugs who do not race across the carpet in full view.

4.3 Routinely use the debugger to follow example execution of code. Can't emphasize this enough. Lookout for the code that worked without having to be debugged in detail!

4.4 Whenever you have a block of new code that you're getting ready to check in, print it out, grab a highlighter, and step through it in the debugger. Highlight every code line that gets hit, and keep using different combinations of inputs until every code line is highlighted. It doesn't take very long, and it's definitely worth it.

4.5 Check for the right number and type of arguments to functions, check the output of functions like `FOPEN, EVAL, FZERO`, etc. for failure (e.g. singularities and other numerical problems) before carrying out the calculations.

Other items:

- Variable precision arithmetic and symbolic math are wonderful features of MATLAB not forget they exist.
- Also remember `EVALIN` and `ASSIGNIN` exist. Using them in a program is probably problematic ☺, but they are very useful for debugging
- "The Practice of Programming" by Brian Kernighan and Rob Pike, Addison-Wesley (1999).
- Steve McConnell, 'Code Complete: A Practical Handbook of Software Construction', Microsoft Press, 1993, ISBN 1-55615-484-4

Compiled by Michael Robbins, CFA; Director, Proprietary Trading; CIBC World Markets Corp; michael.robbins@us.cibc.com; robbins@bloomberg.net