Software-Defined Radio: A Hands-On Course

MATLAB Style, Pointers, and Pitfalls

Last revision: Sept. 16, 2015

Use Constants, Not Literals

- Avoid the use of literals in the code. Instead, define constants at the top of your file.

  Example:

  ```
  ORDER = 2;
  F_SAMPLE = 50000; % Hz
  F_INFO    = 1000; % Hz
  ...
  [b,a] = butter(ORDER, 2*F_INFO / F_SAMPLE);
  ```

  rather than

  ```
  ...
  [b,a] = butter(2, 0.04); % Who remembers what 0.04 meant?
  ```

⟶ This can and *will* save you a lot of errors.

- Do not miss left hand zeros in floating point constants: prone to mistakes

```
threshold = .5;  % Bad: one can easily read 5
threshold = 0.5; % Good
```

Comment Your Code

- Comments not only help someone else to better unterstand your code, but also make it easier for yourself to retrace your thoughts.
- Comments help even when *writing* code, since you can write out what you will do before doing it:

```
% First we'll filter the signal,
% then we recover the message symbols.
...
```

- This is of course useless:

```
% Set a to 5
a = 5;
```

Write Function Descriptions

By writing a description in English of how exactly a function works, it will be much easier to actually write the functions, and it will of course help anyone who later uses the function (including yourself):

```
function a = foo(b, c)
% FOO Short description here
%    A = FOO(B, C) returns the foomatic number of
%    the vectors B and C.
%    If B and C are matrices, the foomatic numbers are
%    computed columnwise.
%
%    A = FOO(B, C, Q) allows you to specify in addition
%    the foomatic parameter Q.
...
```

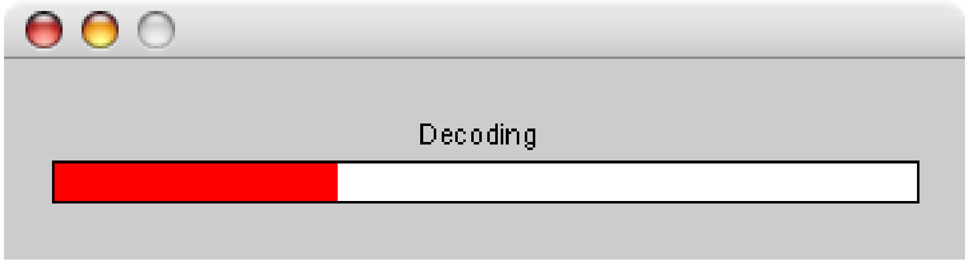
This description will be available if we type `help foo` at Matlab's prompt

Time Consuming Operations

- Your code should display what it's doing if it doesn't otherwise produce any output for several seconds.
- Use `fprintf` to display text, *e.g.*,

```
fprintf('%d percent done\n', k / total_iter);
```

- For a nice graphical display, use `waitbar` (see help):

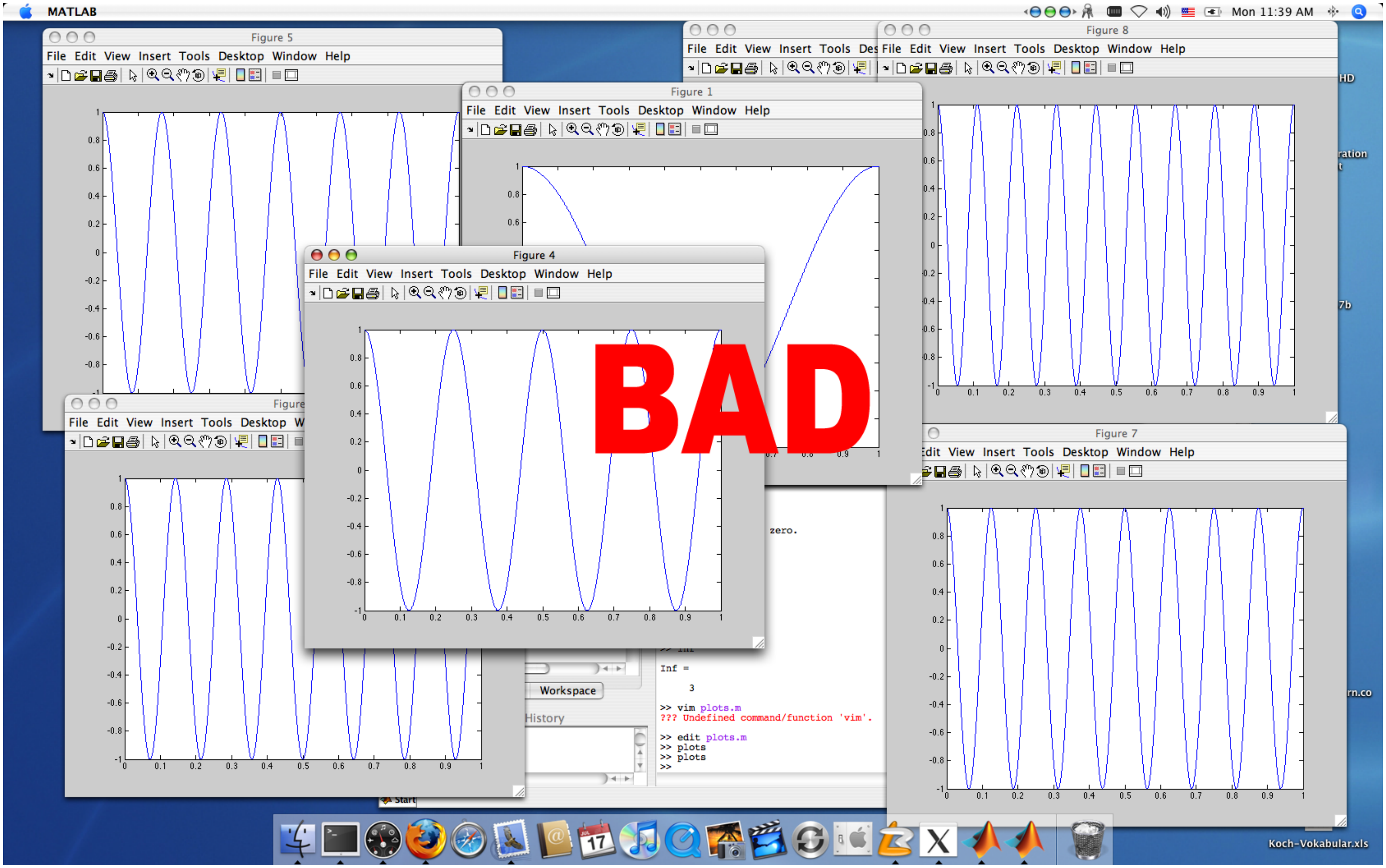


Decoding

Figures

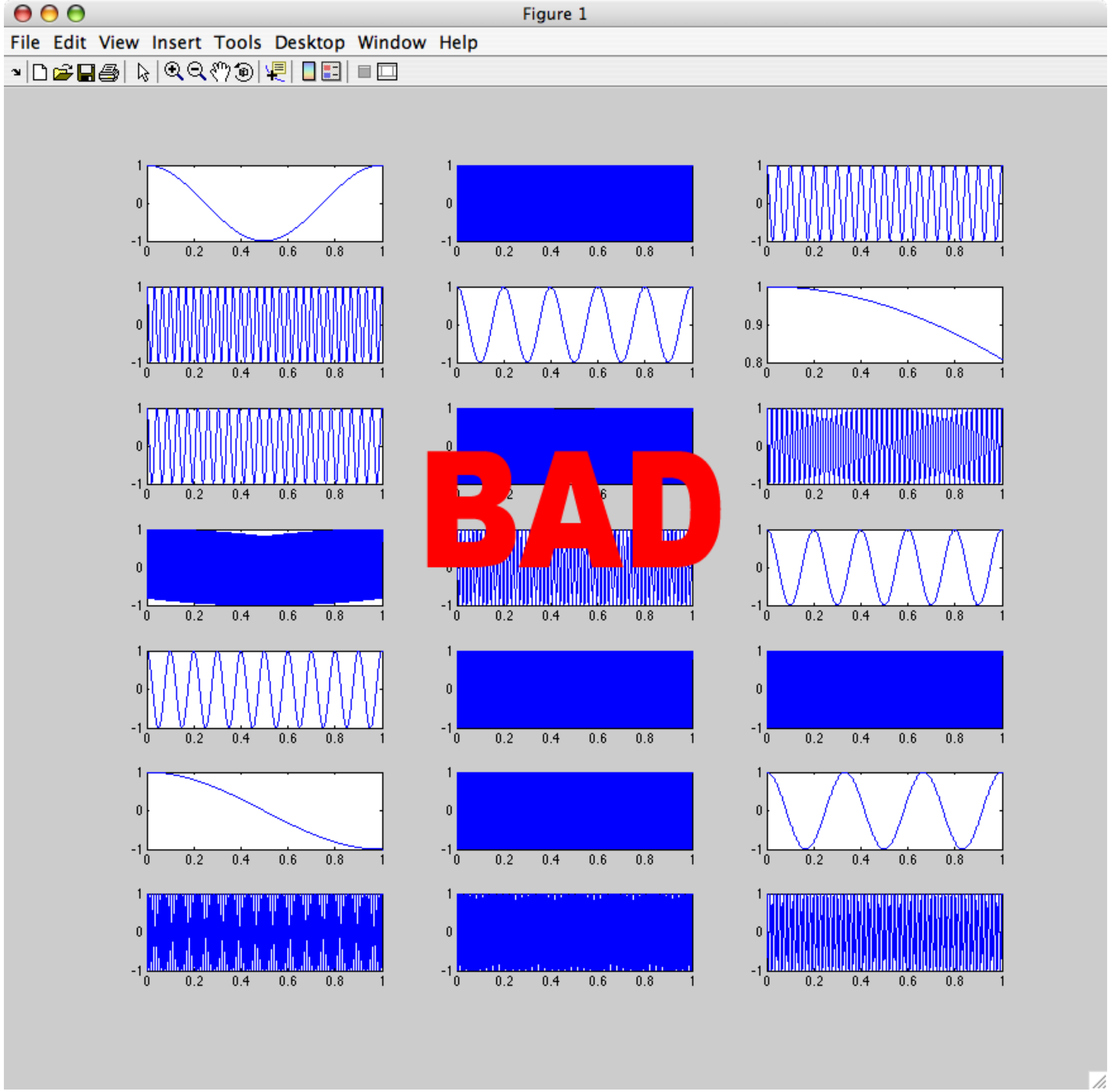
- Use `subplot` when appropriate to avoid cluttering the screen with figure windows
- All plots must have a *title* and *labeled axes*. Use the commands `title`, `xlabel`, and `ylabel`.
- Pictures say more than 1000 words. . .

Figures (cont'd)

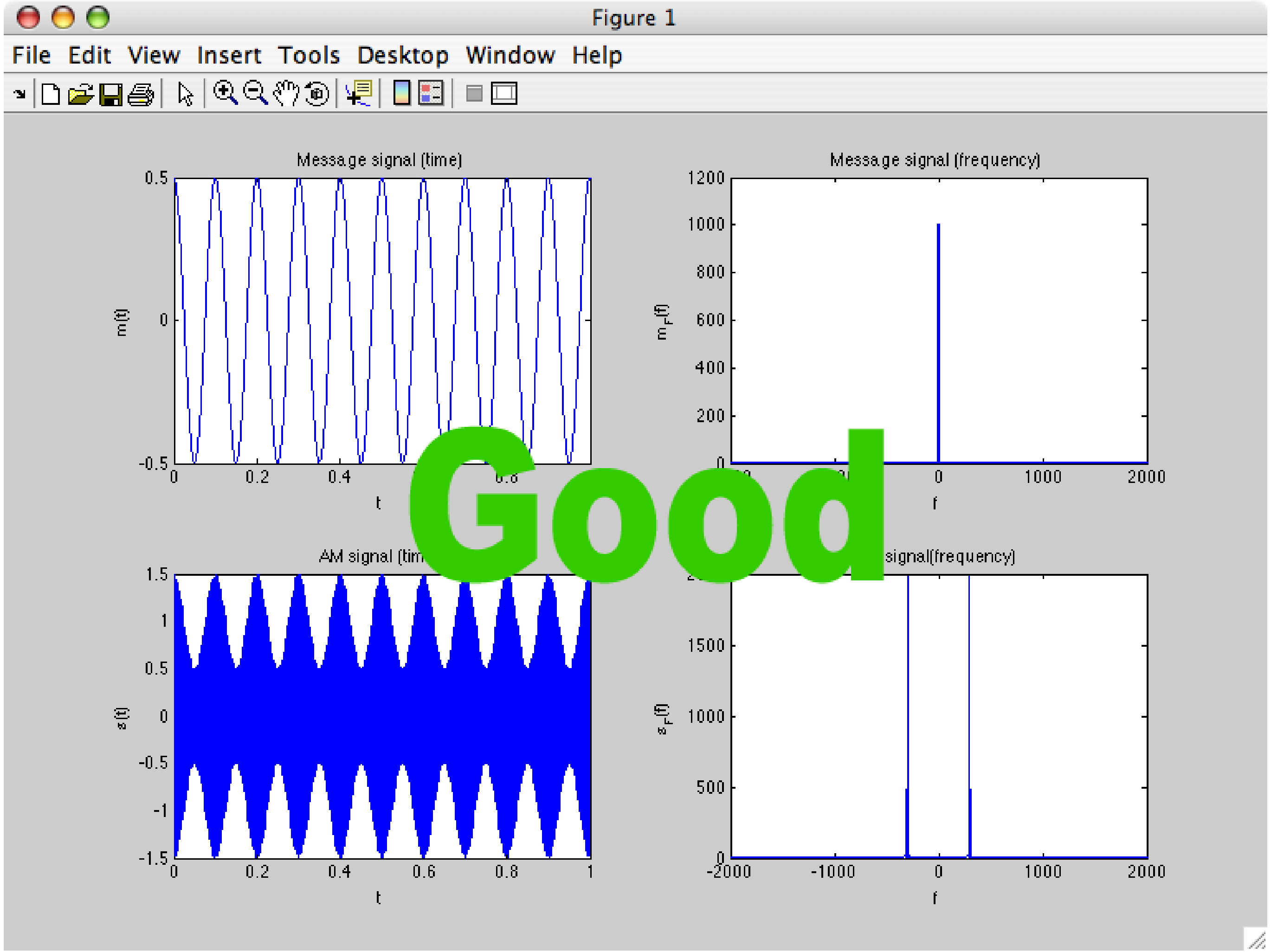


```
zero.

Inf =

     3

>> vim plots.m
??? Undefined command/function 'vim'.

>> edit plots.m
>> plots
>> plots
>>
```

Figures (cont'd)



Figures (cont'd)



Don’t Overwrite Built-In Constants

- MATLAB lets you overwrite built-in constants or function names without warning:

```
pi = 4;   % revolutionize geometry
var = 7;  % we cannot access the function to compute
          % the variance any more
```

- This can lead to errors that are hard to track:

```
for i = 1:length(some_vec)
  ...
  s = exp(i*2*pi*t*f_c);   % not a sinusoid!
  ...
end
```

- If in doubt, use `which -all` *`name`* to check if *`name`* will shadow an already existing function or variable

- Always use `1i` (or `1j`) instead of simply `i` (or `j`) for the imaginary unit. For instance, complex constants can be written as `3+1i*2` or, even better, as `3+2i`.

- (You can also use `j` instead of `i`.)

- As an alternative, you can always type `i = sqrt(-1)` if you have overwritten the imaginary unit.

Why `for`-loops Are Bad in Matlab

- Built-in Matlab functions are optimized to work with matrices (rather than with single numbers)
- Main parts are implemented in `C`
- Using `for`-loops destroys this optimization, since the `C`-functions have to “return” to Matlab after processing each element of a vector

for Loops: A Typical Example

- Problem: Convert uniform random numbers to 0's and 1's
- Bad solution:

```
x = rand(1, NBITS);     % Create uniform random numbers
for k = 1:length(x)
  if x(k) > 0.5
    x(k) = 1;           % Set bit to 1
  else
    x(k) = 0;           % Set bit to 0
  end
end
```

`for` **Loops: A Typical Example (cont'd)**

- Problem: Convert uniform random numbers to 0's and 1's
- Good solution: logical indexing

```
x(x > 0.5) = 1;
x(x <= 0.5) = 0;
```

- Even shorter:

```
x = (x > 0.5);
```

(Possible disadvantage: Data type of `x` now `logical`)
(Possible advantage: one `logical` requires 8 times less memory than a `double`)
Related tip: use command `whos` to know the type and size of your variables in memory

for Loops: Example 2

Application of functions to vectors:

- Bad solution:

```
for k = 1:M
  sym_const(k) = exp(j*2*pi*(k-1)/M);
end
```

- Good solution:

```
sym_const = exp(j*2*pi*[0:M-1]/M);
```

Many Matlab functions do not only accept vectors as input, but matrices, operating in some cases on each column independently (for instance sum, max, min, prod, fft,etc) ⇒
Organize your data appropriately taking this into account to improve performance.

`for` Loops: Example 3

Using one vector as index of another

```
constellation = ...;
% Maps source symbols to constellation symbols
```

- Bad solution:

```
for k = 1:length(dataSymb)
  constSymb(k) = constellation(dataSymb(k));
end
```

- Good solution:

```
% dataSymb used directly as index
constSymb = constellation(dataSymb);
```

`for` **Loops: Conclusions**

Replacing `for`-loops by equivalent matrix operations (often called *vectorizing* your code) makes it

- faster,
- shorter,
- (in most cases) more readable.

This will come in handy in future assignments where you will be dealing with large amounts of data.

Creative use of `find`, `repmat`, `reshape`, `kron`, `bsxfun`, the colon operator (`:`), etc can really help vectorize your code, but try not to make the code too cryptic. Readability is very important.

Preallocation

- Do not make vectors or matrices grow inside `for` or `while` loops
- Preallocate whole memory before filling elements.
  Use commands `zeros`, `ones` for this.
- Example: bad

  ```
  x = 0;
  for k = 2:1000
    x(k) = x(k-1) + 5;
  end
  ```

- Example: good - no need to repeatedly reallocate memory and move data as more values are assigned to `x` in the loop

  ```
  x = zeros(1,1000);
  for k = 2:1000
    x(k) = x(k-1) + 5;
  end
  ```

- Preallocation may not always be possible, $i.e.$, when you do not know the size of the resulting matrix

Odds and ends

- Keep code lines short. Use the ellipsis (`...`) if you need to break a long line). Easier to read, easier to print.
- Although not strictly necessary, make function names match the filename. And remember that case matters.
- Do not mix assigments of real and complex matrices
- Do not change variable's data types

```
x  = zeros(1, 1E4)
% -- other code --
x =  'ABC';
```

The change of `x` from `double` to `char` has a negative impact on performance.
When you need to store data of a different type, create a new variable.

- Use appropriate logical operators: `&` is not the same as `&&` (short circuit form)
- Pay attention to the use of Hermitian transpose (`A'`) and plain transpose (`A.'`)
- Remember: indices in Matlab are 1-based (as opposed to C, Python, Java ..., where array indices are 0-based)
- Use Matlab tools: debugger, profiler, variable inspector
- Make your code modular. Break complex functions into simpler functions. If one function is only meant to be called by another, then make it a subfunction of that one.
- Use scripts carefully. They are slower than functions, and you are mixing the command-line workspace with that of the script (prone to conflicts)

Reminder

- The guidelines presented here are not only a suggestion, we will also consider their correct application by you when grading.
- Bad style ⟶ bad (or not so good) grade :(

Recommended reading / reference:

- Ed Overman, "*A MATLAB Tutorial*", Department of Mathematics, Ohio State University.
  Available on Moodle.

Go through this tutorial as deeply as you think you need as part of the homework assignment.

⟶ Questions?