Advanced Computing Tools for Applied Research

Chapter 2. Coding conventions

Jaime Boal Martín-Larrauri
Rafael Palacios Hielscher

Academic year 2014/2015

Why should you bother about coding conventions?

Coding conventions are a set of rules or guidelines used when writing source code for a computer program

- Code should be written with others in mind
  - You will rarely write code just for yourself ➡ Most projects are collaborative
  - Hardly any software is maintained for its whole life by the original author
  - Therefore code should be easy to read (by human beings, not machines)
  - Coding conventions allow programmers to understand new code more quickly and thoroughly
  - When you go through a code snippet you wrote several weeks ago it will be completely new to your eyes ➡ Believe me, it happens

Disclaimer

- Guidelines are not commandments
  - Any violation is acceptable if it enhances readability
  - If you have strong personal objections against a rule, share your thoughts with others and see if you can come up with a better alternative
- This presentation does not intend to be a comprehensive enumeration of conventions, but highlight the most widespread standards
- MATLAB notation will be employed for examples without loss of generality
- Every development team or organization might have their own, perfectly valid, coding style
  - Ask your team if they already follow some guidelines and stick to them
  - Feel free to adopt these ones otherwise!

General conventions

1. Everything should be written in English
   - Variable and function names, comments…
   - English is the preferred language for international development

```
length                      % NOT: longitud, longueur...
computeArea();              % NOT: calcularArea(), calculerSurface()...
```

2. Abbreviations in names should be avoided

```
averageTime                 % NOT: avgTime
initializeAlgorithm();      % NOT: initAlgorithm();
```

   - Domain specific phrases that are more naturally known through their acronym should be kept abbreviated

```
cpuFrequency                % NOT: centralProcessingUnitFreq
convertToHtml();            % NOT: convertToHypertextMarkupLanguage();
```

3. Use parentheses to clarify expressions and indicate precedence

Naming conventions | Variables or attributes

4. The names of variables should document their meaning or use

```
area = length * width;  % NOT: a = l * w;
```

5. Variable names should be in mixed case starting with lower case
   - Another alternative is to separate words using underscores

```
vehicle                % NOT: Vehicle, VEHICLE...
electricVehicle        % NOT: electricvehicle, Electricvehicle...
electric_vehicle       % Less common but still used
```

6. Abbreviations and acronyms, even if normally uppercase, should be mixed or lower case when used as a name

```
html                   % NOT: HTML, hTML
isUsbDetected          % NOT: isUSBDetected
```

7. Dimensioned variables (and constants) should have a units suffix

```
incidentAngleRadians   % NOT: incidentAngle
```

Naming conventions | Variables or attributes

8. Variables with a large scope should have meaningful names, variables with a small scope can have a short name
   - Scratch variables used for temporary storage or indices are best kept short

```
i, j, k, m, n             % Integers (Note that l is avoided)
c, d                      % Characters
x, y, z                   % Floating point numbers
```

9. Iterator variables should be named or prefixed with *i*, *j*, *k* if the loop is rather small (e.g., less than 10 lines). For larger loops it makes sense to use names sufficed *idx*
   - For nested loops the iterator variables should be in alphabetical order

```
for iFile = 1 : nFiles
  for jRow = 1 : nRows
    ...
  end
end
```

Naming conventions | Variables or attributes

10. The prefix *n* should be used for variables representing the number of objects

```
nObjects                    % Better than: numberOfObjects
```

11. Variables representing a single entity number can be suffixed by *No* or prefixed by *i*

```
tableNo, employeeNo
iTable, iEmployee           % Variables become named iterators
```

12. Plural form should be used on names representing a collection of objects
    - A single convention on pluralization should be followed consistently

```
pointArray                  % NOT: points (can be confused with a variable)
```

Naming conventions | Variables or attributes

13. The prefix is should be used for boolean variables (and functions)

```
isSet, isVisible, isFinished, isFound, isOpen
```

- In some cases *has*, *can*, and *should* prefixes might be better alternatives

```
hasLicense, canEvaluate, shouldSort
```

14. Negated boolean variable names should be avoided
    - Can lead to confusion when used in an expression

```
isError                     % NOT: isNoError
isFound                     % NOT: isNotFound
```

15. Variables should be initialized when they are declared
    - If this is not possible, they should be left uninitialized rather than initialize them to a phony value

Naming conventions | Constants

16. Named constants should be all uppercase using underscores to separate words

```
MAX_ITERATIONS, PI
```

17. Constants can be prefixed by a common type name

```
COLOR_RED, COLOR_GREEN, COLOR_BLUE
```

18. Floating point constants should always be written with
    - Decimal point and at least one decimal
    - A digit before the decimal point

```
total = 5.0                % NOT: total = 5.
total = 0.5                % NOT: total = .5
```

19. The use of magic numbers in the code should be avoided
    - Numbers other than 0 and 1 should be considered declared as named constants instead

Naming conventions | Functions or methods

20. Functions names should document their use

```
computeWidth();                     % NOT: compwid()
```

21. Names representing functions or methods must be verbs and written in mixed case starting with lower case

```
getName();                          % NOT: GetName(), getname(), ...
computeTotalWidth();                % NOT: totalWidth(), total_width(), ...
```

22. Functions with no output argument or which only return a handle should be named after what they do

```
draw(), plot(), add(), multiply(), read();
```

23. The terms *get/set* should be used where an attribute is accessed directly

```
getTemperature(), setPassword();
```

Naming conventions | Functions or methods

24. The terms is (or *has*, *can*, or *should* when appropriate) should be used for boolean functions

```
isEmpty(), isFull(), hasLicense(), canEvaluate(), shouldSort();
```

25. The term compute can be used in methods where something is computed
    - Gives an immediate clue that it is a potentially complex or time-consuming operation

```
computeInverse(), computeProbabilities();
```

26. The term *find* can be used in methods where something is looked up
    - Gives an immediate clue that it is a simple look up method with a minimum of computations involved

```
findLastRecord(), findMaximum();
```

Naming conventions | Functions or methods

27. The terms *initialize* can be used where an object or a concept is established
    - American *initialize* should be preferred over British *initialise*. Refrain from using the abbreviation *init*.

```
initializeAlgorithm(), initializeCamera();
```

28. Complement names should be used for complement operations

```
add/remove              insert/delete           open/close
begin/end               increment/decrement     up/down
create/destroy          min/max                 show/hide
first/last              next/previous           start/stop
get/set                 old/new                 suspend/resume
```

Statements | Conditionals

29. Complex conditional expressions should be avoided
    - Introduce temporary logical variables instead

```
% NOT:
if (value >= lowerLimit) & (value <= upperLimit) &
    ~ismember(value, valueArray)
  ...
end

% Use:
isWithinRange = (value >= lowerLimit) & (value <= upperLimit);
isNew         = ~ismember(value, valueArray);

if (isWithinRange & isNew)
  ...
end
```

Statements | Conditionals

30. In general, the usual case should be put in the *if* clause and the exception in the *else* part of an *if-else* statement

```
fileId = fopen(fileName);

if (fileId ~= -1)
  ...
else
  ...
end
```

31. Implicit test for `0` should not be used other than for boolean variables (and pointers)

```
if (isOk)                % Better than: if (isOk ~= 0)
if (nElements ~= 0)      % NOT: if (nElements)
if (value ~= 0.0)        % NOT: if (value)
```

Layout

32. Content should be kept within the first 80 columns

33. Lines should be split at graceful points
    - Break after a comma, a space, or an operator
    - Align the new line with the beginning of the expression of the previous line

```
totalSum = a + b + c + ...
           d + e;
function(param1, param2, ...
         param3);
```

34. Indentation enhances readability
    - In some languages like Python it is even meaningful!

35. Use spaces for indentation rather than tabs
    - Use 2 to favor less line splits
    - Use 3 or 4 to emphasize the logical layout of the code

Layout

36. Conventional operators should be surrounded by a space
37. Reserved words should be followed by a space
38. Commas should be followed by a space
39. Colons should be surrounded by space
40. Semicolons within statements (not at the end of line) should be followed by a space character

```
a = (b + c) * d;            % NOT: a=(b+c)*d
if (isOk)                   % NOT: if(isOk)
doSomething(a, b, c, d);    % NOT: doSomething(a,b,c,d);
for i = 1 : nRows           % NOT: for i = 1:nRows
a = [1 2; 3 4];             % NOT: a = [1 2;3 4];
```

Layout

41. Short single *if*, *for*, or *while* statements can be written in one line

```
if (condition), statement; end
while (condition), statement; end
for i = 1 : nElements, statement; end
```

42. Logical units within a block should be separated by a blank line

43. Functions should be separated by three blank lines

44. Use alignment wherever it enhances readability

```
width  = getWidth();
height = getHeight();

area      = computeArea(width, height);
perimeter = computePerimeter(width, height);
```

Comments

- More on this in Chapter 3

45. Tricky code should not be commented but rewritten!

46. Comments should agree with the code, but do more than restate it

47. Comments should be easy to read
    - They should start with an upper case letter and end with a period

48. Comments should usually have the same indentation as the statements referred to

49. Function header comments should discuss any special requirements for the input arguments

50. Function header comments should describe any side effects

1

MATLAB

Naming conventions | Variables or attributes

8. Iterator variables should be named or prefixed with *i*, *j*, *k* if the loop is rather small (e.g., less than 10 lines). For larger loops it makes sense to use names sufficed *idx*
   - Applications using complex numbers should reserve *i*, *j* or both for use as the imaginary number

10. The prefix *n* should be used for variables representing the number of objects
    - Use *m* for the number of rows ➡ Based on matrix notation

```
mRows                    % MATLAB uses nrows
nColumns                 % MATLAB uses ncols
```

- Structure names should begin with a capital letter
  - The name of the structure is implicit and need not be included in the fieldname

```
Segment.length           % NOT: Segment.segmentLength
```

Naming conventions | Functions or methods

20. Names representing functions or methods must be verbs and written in mixed case starting with lower case
    - Most MATLAB built-in functions are written in lower case
    - Formerly, using lowercase avoided potential file name problems in mixed operating systems environments ➡ Not an issue anymore

- Functions with a single output can be named after the output
    - Common practice, but potentially dangerous ➡ Avoid it if you can

```
mean(), median(), std(), min(), max();
```

- In MATLAB some “constants” use lowercase names (e.g., pi)
    - Built-in constants are actually functions

- It is clearest to have the function and its m-file names the same

2

GAMS

Naming conventions

- GAMS is **case insensitive** (but who knows if this may ever change)

4. Variables should document their meaning or use
   - Keep names short without compromising this convention to enhance readability ➡ Long names do not work well in multicolumn displays

5. Variable names should be in mixed case starting with lower case
   - Extend this convention to parameters and equations
   - Let parameters start with *p*, variables with *v*, and equations with *e*

```
p_windSpeed                 !! Better than: pWindSpeed
v_energyGenerated           !! Better than: vEnergyGenerated
e_loadBalance               !! Better than: eLoadBalance
```

- Use one or two letter set names

```
SET   h   hour   / h-0 * h-23 /
```

Layout

- GAMS was designed to include documentation within the code
  - One declaration per line is recommended since it encourages commenting

```
SCALARS
p_batteryCapacity       Battery capacity [kWh]    /  24 /
p_nBatteries            Number of batteries [-]   / 500 /
;

PARAMETERS
p_windSpeed(h)          Wind speed at hour h [m/s]
p_solarIrradiance(h)    Solar irradiance at hour h [W/m^2]
;

VARIABLES
v_energyCharged(h)      Energy charged at hour h [kWh]
v_energyDischarged(h)   Energy discharged at hour h [kWh]
;
```

References

- S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd Ed., Microsoft Press, Redmond, WA, USA, 2004.
- R. Johnson, *MATLAB Style Guidelines* 2.0, Mar. 2014. www.mathworks.com/matlabcentral/fileexchange/46056-matlab-style-guidelines-2-0
- B. A. McCarl, *Good Modeling Practices: Enhancing GAMS Model Self Documentation*, 2001. http://www.gams.com/mccarl/goodmodl.pdf
- W. Britz, The red book on CAPRI GAMS coding, University of Bonn, Feb. 2010. http://www.ilr.uni-bonn.de/agpo/rsrch/capri-rd/docs/d7.1.pdf
- A. Ramos, *Look and Feel of StarGen and StarNet Lite Models*, Nov. 2014. http://www.iit.upcomillas.es/aramos/presentaciones/StarLite_LookAndFeel.pdf
- Google, Inc., *Google C++ Style Guide*, 2014. google-styleguide.googlecode.com/svn/trunk/cppguide.html
- T. Hoff, *C++ Coding Standard*, 2008. www.possibility.com/Cpp/CppCodingStandard.html
- M. Henricson and E. Nyquist, *Programming in C++, Rules and Recommendations*, 1992. www.doc.ic.ac.uk/lab/cplus/c%2b%2b.rules/