Matlab coding standard for Autonomous agents, FFR125 v 1.1, 2009-02-02, v 1.2, 2009-08-25, v1.3 2011-01-11

David Sandberg, Mattias Wahde

1 Introduction

When writing code, in most cases you are not only writing for the computer to understand and make use of your code, but people as well! This, of course, includes yourself. Will you understand your code a month after having written it? By writing clear and highly readable code you reduce the risk of introducing unwanted errors. It is the aim of this coding standard to help you write such code.

Note that when solving home problems involving programming, you *should* use Matlab, and you *should* follow the code standard described below. Programs that deviate significantly from the code standard will result in deduction of points *even* if the code works as intended.

2 Naming practices

When naming a variable (or structure, method global constant etc.) you should always strive to use a *meaningful* name that clearly describes the purpose of the variable. For example a good name for the variable used to store the number of individuals in a population is `populationSize` whereas simply naming the variable `n` is not recommended. Using long names is perfectly acceptable; When given a choice, a long but descriptive variable name is to be preferred over a short name with unclear meaning.

2.1 Variables

For variable names, the first character should be in lower case. If the variable name consists of several words, all words except the first should begin with an upper case letter, and all other letters should be lower case, e.g. `aLocalVariable`. As stated above, using meaningful names should always have higher priority than using short names and this is especially so in the case of variables with a large scope. However, variables with limited scope can have short names. For example, a variable used for storing a temporary value within an `if-then` statement (containing only a few lines of code between `if(...)` and `end`) may very well be named `tmpVal`. In the case of a real-valued variable simply naming the variable `x` suffices.

2.1.1 Iterator variables

Iterator variables should be named using `i`, `j` and `k`. However, in the case of a loop consisting of many lines of code, a longer and more meaningful name should preferably be used and should be prefixed with either `i`, `j` or `k`. As an example, consider the `iGeneration` iterator variable of the main loop in `FunctionOptimization.m`.

2.1.2 Abbreviations

Abbreviations in variable names (as well as in names for methods etc.) are acceptable in the case of very common abbreviations, e.g. max and html. Note that the upper and lower case rule above still applies, e.g. `cthStudent` *not* `CTHStudent`.

2.1.3 Prefixes

As noted above, when using a longer name for an iterator variable, the name should include a prefix such as `i`. Also, the same prefix should be used for indexing variables, e.g. `iBestIndividual` in `FunctionOptimization.m`. In the case of a variable storing an integer quantity, e.g. the number of genes in a chromosome, the name of that variable should include the prefix `n`, i.e. `nGenes` (`nrOfGenes`, is also ok).

2.2 Functions

Functions should be named using upper case for the first character of *every* word in the function name, i.e. `InitializePopulation`. Function parameters are named as variables. While variables are often named using nouns, function names should preferably include at least one verb, since a function is intended to perform some action. The file name should match the function name. For example, the file containing the function `InitializePopulation` should be named `InitializePopulation.m`.

2.3 Structs

The code generating a struct is named using uppercase letters the for the first character of each word in the name, i.e. using the same standard as for function names. The fields (defining the struct) are given names in uppercase letters. The corresponding variables begin with lowercase letters, i.e. using the same principles as for variables in general, see above. Example:

```
function c = CreateCompass(name,epsilon,sigma);

c = struct('Name',name,...
           'Type','Compass',...
           'EstimatedHeading', 0,...
           'Epsilon',epsilon,...
           'Sigma',sigma);
```

Note that, when a struct is used, it should be written with an initial lowercase letter (i.e. using the same principles as for variable in general, see above). Example:

```
epsilon = 0.0050;
sigma = 1.0000;
compass = CreateCompass('compass',epsilon,sigma);
```

2.4 Global variables and constants

A global variable (declared using the keyword `global`) should be named using only capital letters, and with _ in between words, e.g. `A_GLOBAL_VARIABLE`. However, the use of global variables and constants should be kept to a minimum.

3 Code organization and layout

3.1 Whitespace and other layout topics

Use whitespace to group your code in order to make it more readable. Use vertical whitespace (i.e. blank lines) to form blocks of lines of code, quite similar to paragraphs when writing normal text. Note that the lines of code that constitute a block should be cohesive (i.e. the lines of code should be strongly related to each other) and the formation of the block should be logical. As an example, study the file `FunctionOptimization.m`. Use horizontal whitespace (i.e. indentation) to group statements, such as `if-then-else` and `for`-loops. Use two blank spaces for indentation, on the form given in the following example.

```
if (r < crossoverProbability)
  newIndividual = NewIndividualsByCrossover(population,i1,i2,nGenes);
  temporaryPopulation(i,:) = newIndividual(1,:);
  temporaryPopulation(i+1,:) = newIndividual(2,:);
else
  temporaryPopulation(i,:) = population(i1,:);
  temporaryPopulation(i+1,:) = population(i2,:);
end
```

Note the use of the two temporary variables `i1` and `i2` in the example. This is perfectly fine as the scope is small. Furthermore, since these variables are used for indexing, they are prefixed with `i`.

3.2 Avoid complex statements

In order to improve readability and avoid errors in code, avoid writing statements that each perform many steps of computation. For example, the code snippet

```
result = (abs(timeSeries.Value(i) - (delta*avg + (1-delta)*gamma))^kappa)* ...
         weightLeft/ (1+exp(alphaLeft*(timeSeries.Value(i) - (delta*avg + ...
         (1-delta)*gamma)-betaLeft)))+weightRight/(1+exp(-alphaRight* ...
         (timeSeries.Value(i) - (delta*avg + (1-delta)*gamma)-betaRight)));
```

should be written using several statements and temporary variables

```
x = timeSeries.Value(i);
z = x - (delta*avg + (1-delta)*gamma);
wR = weightRight/(1+exp(-alphaRight*(z-betaRight)));
wL = weightLeft/(1+exp(alphaLeft*(z-betaLeft)));
w = wR+wL;
result = (abs(z)^kappa)*w;
```

3.3 Conditional expressions

Avoid complex conditional expressions spanning over several lines. Instead, introduce temporary boolean variables.

```
if (not(location == BUNKER) && ((sustainedWinds == CATEGORY_2_SUSTAINED_WINDS)
   &&(centralPressure == CATEGORY_2_CENTRAL_PRESSURE)))
  ...
end
```

should instead be coded as

```
isCategory2Winds = (sustainedWinds == CATEGORY_2_SUSTAINED_WINDS);
isCategory2Pressure = (centralPressure == CATEGORY_2_CENTRAL_PRESSURE);
isCategor2Hurricane = (isCategory2Winds && isCategory2Pressure);
outsideBunker = not(location == BUNKER);
if (isCategory2Hurricane && outsideBunker)
  ...
end
```

4 Code optimization

Whenever you have the choice between (1) clear but slower code or (2) cryptic and perhaps faster code you should always choose the clear and more readable code. Do not bother with trying to improve the performance of your code by vectorization. However, this does not imply that you should not pay attention to the speed performance of your code. In order to improve the speed performance of loops you should, for example, always initialize vectors and matrices before the loop, i.e.

```
fitness = zeros(populationSize);
for i = 1:populationSize
  parameterValues = DecodeChromosome(population,i,range,nGenes);
  fitness(i) = 1.0/EvaluateIndividual(parameterValues);
  ...
```

5 Comments

You should strive to write code that is self explanatory. However, sometimes it is needed to add information to the code, such as an explanation of a complex algorithm, information regarding limits or perhaps a motivation. Add such information in comments, using (%), and try to do so at the time of writing the code. However, do not overuse comments! Including unnecessary comments such as

```
i = 1; % Assigns 1 to the variable i
```

is not a good idea.

6 Program output

In general, Matlab will print the result of (for example) a function call or an assignment, unless the line ends with a semi-colon (;). In some case, particularly when debugging, it can be a good idea to print out quite a lot of information. However, in the final version of a program, only *relevant* information should be printed, and only if the information can be well represented in text format. Thus, for example, a program for function optimization might print the best function value found, as well as the corresponding variable values, every n^{th} generation, where n should be sufficiently large such that the information will appear in the Matlab main window with a reasonable frequency (i.e. not more than once per second, or so). It is *not* good programming practice to dump excessive amounts of irrelevant information (or information that cannot be analyzed in real time) to the screen such as, for example, the entire genome of every individual evaluated in an EA. In some cases, for example certain applications of image processing, a text-based representation is not very useful. If, for example, a Matlab program generates an image by manipulating an input image, there is no point to print the RGB values for each pixel to the screen. Instead, the program might, for instance, display the input image and the processed image.