Problem set 2a: M-file Programming

Handed out: Friday, October 7, 2016.

Due: 11:55pm, Friday, October 14, 2016.

You must hand in two files:

- One file, named your_name_E2a.pdf, with the solutions to the exercises in this set, following the template available in the virtual campus. For every exercise, you must explain how you solved it, the necessary MATLAB code, the results obtained (run transcripts or plots), and some final comments (if any).
- The second file, named your_name_E2a.zip, must contain all the MATLAB code you used to solve the exercises, organized in one or more text (.m) files, to allow the teacher to check your solution.

Please, make sure that the code in the pdf file is **exactly the same** as in the zip file. Remember that all the MATLAB code is supposed to be **entirely yours**. Otherwise, you must clearly specify which parts are not, and properly attribute them to the source (names of collaborators, links to webpages, book references,...). Read the Course Information document for more details on this subject.

In those exercises where a function is implemented, a "help" section must be included in the software so that when the Matlab command help is called with the name of that function proper information is provided.

Comment: In this unit it is assumed that you already know how to operate vector and matrix variables. The goal in this exercises is to implement the requested task as efficiently as possible, considering that efficiency is mostly related to the software execution time. In general, Matlab programs should be written avoiding as much as possible the use of loops, resorting to vectorization. As a rule of thumb, the shorter the Matlab program file is the faster it will execute. Although the use of specific data types might improve the efficiency of some exercises in this assignment, this question is beyond the scope of this unit.

Exercise 1. Evaluation of simulation time

The matlab file "exec_time.m" evaluates the execution time of several implementations of a matrix product. You are asked to run the script, discuss the results and draw conclusions on what implementation is more efficient.

Exercise 2. Numerical integration

The Simpson's rule is employed to evaluate the definite integral of a function. Let's consider that we know the value of the function $\boldsymbol{f(x)}$ at an odd number of uniformly spaced values $\boldsymbol{x_j = a + j\Delta, j = 0, \dots, n}$ with $\Delta = \boldsymbol{(b-a)/n}$, so $\boldsymbol{x_o = a}$ and $\boldsymbol{x_n = b}$. Then the definite integral can be approximated as:

$$\int_a^b f(x)dx \approx \frac{\Delta}{3}[f(x_o) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n)]$$

We want to assess the quality of this numerical method by evaluating the differential entropy of a random variable with Laplace distribution with parameter (μ, b):

$$f(x) = \frac{1}{2b}e^{-|x-\mu|/b} \qquad -\infty < x < \infty$$

The differential entropy is defined as

$$H = -\int_{-\infty}^{\infty} f(x) \ln f(x)\ dx$$

and in the case of Laplacian random variables it is equal $1 + \ln(2b)$ nats. You are asked to:

1) Create a function that takes as input the vector with samples $(x_o \ldots x_n)$ and the vector with the function values at those samples $(f(x_o) \ldots f(x_n))$ and returns as output the definite integral of that function with x_n and x_o as upper and lower limits respectively. The function must check if the samples $x_o \ldots x_n$ are equispaced and if the number of samples is odd; if any of these two conditions is not verified then an error message should be displayed and execution should be aborted.

2) Create a function that takes as input the vector with samples $(x_o \ldots x_n)$ and returns the Laplace probability density function evaluated at those values of the argument x (`exp, abs`).

3) Write a script to call these functions and evaluate the differential entropy of the Laplacian distribution with parameters $(2,4)$ by choosing suitable values of x_o, x_n and Δ (`log`). Assess the precision of this procedure to evaluate the differential entropy numerically. S*uggestion:* you may plot the argument in the entropy integral to decide the values of of x_o, x_n and Δ.

Exercise 3. Mathematical equation.

Create a local function to implement the following expression (**`inline`**):

$$g(x) = 1 - e^{-bx}\cos(ax)$$

where a, b are function parameters.

Exercise 4. Search for local minima of a function

The file "ex4_data.mat" contains the samples of a function $f(x)$: vector `x` contains a list of the x-values where the function has been evaluated and vector `f` contains the corresponding set of function values.

Write the script to:

1) Load the file in the matlab worskpace (**`load`**)

2) Search for all the local minima of the function in the file, defining as a minimum of the function the values that are smaller than the samples that precede it and follow it (that is, the first and last sample cannot be local minima) (**`length`**, **`>,<,find`**)

3) Plot the original function along with stars in the locations of the minima (**`plot`**, **`hold on/off).`**

Comment: In section 2 the Matlab command **`local_max`** could be useful; as a learning exercise on indexing and flow control you are requested not to use it.

Exercise 5. Run length encoding

Run-length encoding is a simple compression scheme that is useful when long bursts of 0's and 1's appear in the binary source to be encoded. It consists on storing the number of consecutive 0's and 1's rather than the bits themselves. For example, we could encode the sequence

0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0

as

8 '0's, 4 '1's, 7 '0's, 7 '1's, 2 '0's, 1 '1's, 5 '0's

In this example we could encode each integer with 3 bits (length values in the range 1 to 8) plus one extra bit at the beginning to indicate that the sequence starts with a 0, so we would use 22 bits to compress a sequence of 34 bits. Note that, as the number 0 is never encountered (bursts of length 0 do not exist) we can employ the binary tuples 000 to 111 to represent the numbers 1 to 8. In this example the output of the encoder would be (most significant bit at the left would be) :

0 111 011 110 110 001 000 100

You are asked to write the software required to implement this run-length-coding scheme and apply it to the data stored in the variable '`sequence`' stored in the file 'ex5_data.mat'. You are asked to:

1) Write a function named `'d2b_conversion'` that does decimal-to-binary conversion. The function must take as input parameters:

- a vector of non-negative integer numbers (`'int_list'`)
- the number of bits to employed to represent each integer number with a binary tuple (`'n_bits'`).

and must return as output parameter:

- A vector that contains the stacking of the n_bits-tuples that correspond to the binary representation of the integer numbers (`'bin_list'`)

With tuples of length `n_bits` only the integers in the range 1 to 2^{n_bits} can be respresented. If any of the integer numbers at the input is outside this range an error message should be displayed and the execution should be aborted.

2) Write a function that takes as input parameters:

- the sequence to be compressed stored in a vector (`'sequence'`)
- the number of bits to encode the length of each burst of 0's or 1's into a binary tuple (`'n_bits'`).

and returns as output parameters:

- The value of the first bit
- A vector that contains the length of the bursts of 0's and 1's as the in the example above.
- A vector that contains the binary compressed sequence (that is, a binary sequence with the value of the first bit plus `n_bits` additional bits for each burst of 0's and 1's).

3) Write a script that loads the file "ex5_data.mat" in the workspace and displays in the screen the run-length encoded sequence when 3 bits are used to represent the length of each burst.

Comment: In section 1 the **`de2bi`** command could be useful; as a learning exercise on indexing and flow control you are asked not to use it.

Useful matlab commands: **`load`**, **`floor`** or **`fix`**, **`rem`** or **`mod,==,~=,>, find,isempty.`**