Lab 1: Working with Data

Due date: Tuesday, April 3 (beginning of lab period).

Lab Assignment

In this course you will write a lot of programs that work with large quantities of data. Two forms of data your programs will have to handle most often are CSV files (containing mostly, although not exclusively, numbers) and text files containing English text. This lab asks you to write code for reading, parsing and performing simple manipulations with the data parsed from such files. While not every single piece of code you write for this lab may wind up being useful elsewhere in the course, *some will*.

This is a *pair programming assignment*. You pick your partner during the March 27 lab session. I **strongly discourage** individual work for this (and other team/pair programming) lab(s), even if you think you can do it all by yourself.

Tasks

Data Formats

For this assignment you will work with data stored in two different formats: numeric CSV files and plain text files. The formats are briefly described below.

CSV Files. CSV, or *Comma-Separated Value* file format is one of the most commonly used formats for relational/spreadsheet-style data data transfer. Most of data sets you will be using this quarter will come in this format.

For this lab, you will be working with *numeric* CSV files, i.e., files that store only numbers (integers and/or reals). A CSV file consists of a number of lines. Each line contains a list of values: integers or reals, in our case. The values are separated by commas (','). Missing values are allowed: they are identified by two consecutive commas. Whitespace (spaces, tabs) is ignored, but line breaks signal beginning of a new line (row) of values.

Different lines from the same file are expected to have the same number of values in them, but this is not a guarantee. Sometimes, a single CSV file may contain more than one data table in it, in which case, rows from different tables can have drastically different numbers of values.

Some examples of data in CSV format are:

```
1,  2,  3,  5
10, 11, 12, 14
20, 30, 40, 50
```

In the example above, the dataset consists of three lines (rows), each of which contains four values (columns). All values are integer.

```
1,2,3,4,5,6,7
1,2,3,4,5
1,1,1,1,1,1
```

This dataset contains three rows, but each row has different number of values (columns). All values are integer.

```
1.4, 4.56, 2, 3.45
20.0, 2.2, 3, 4.55
```

Here, we have two rows, each containing four columns. Third column contains integer values, other columns - real values.

```
1.2, 3,, 3.4, 5
5.5,, 2, 3.3, 7
,,,,6
```

This dataset describes three rows, each with five columns. However, not all values are defined. In the first row, the value for the third column is missing, in the second row – the value for the second column and in the third row, all but the last columns have undefined values.

Plain text. Plain text file format is the simplest file format for storage and transfer of textual information. In this lab, we treat the contents of a such files as a text document written in English (the latter, generally speaking, is irrelevant). We assume that plain text files contain only ASCII characters (no need for Unicode support in this lab).

A plain text file is a collection of words that are united into sentences (sequences of words that terminate with one of the specified punctuation signs). Sentences are formed into paragraphs (sequences of sentences that are terminated in an empty line). Words are separated from each other by punctuation, whitespace and line breaks (we assume that no words are split by a line break in the middle). A paragraph break terminates a sentence even if there is no punctuation at the end of it.

Some examples of plain text data:

```
Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.
```

The text above[1] is a single paragraph, consisting of a single sentence.

```
Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of
lather on which a mirror and a razor lay crossed. A yellow dressinggown,
ungirdled, was sustained gently behind him on the mild morning air.
```

This text[2] has a single paragraph with two sentences.

```
Whether I shall turn out to be the hero of my own life, or whether that station
will be held by anybody else, these pages must show. To begin my life with
the beginning of my life, I record that I was born (as I have been informed
and believe) on a Friday, at twelve o'clock at night. It was remarked that
the clock began to strike, and I began to cry, simultaneously.

In consideration of the day and hour of my birth, it was declared by the
nurse, and by some sage women in the neighbourhood who had taken a lively
interest in me several months before there was any possibility of our becoming
personally acquainted, first, that I was destined to be unlucky in life; and
secondly, that I was privileged to see ghosts and spirits; both
these gifts inevitably attaching, as they believed, to all unlucky infants of
either gender, born towards the small hours on a Friday night.
```

This text[3], contains two paragraphs.

```
'That is all right,' said the Psychologist.

'Nor, having only length, breadth, and thickness, can a cube have a real existence.'

'There I object,' said Filby. 'Of course a solid body may exist. All real things'
```

[1] Lewis Carroll, Jabberwocky.

[2] James Joyce, Ulysses, http://www.gutenberg.org/files/4300/4300-h/4300-h.htm

[3] Charles Dickens, David Copperfield, http://www.gutenberg.org/files/766/766-h/766-h.htm

`'So most people think. But wait a moment. Can an instantaneous cube exist?'`

`'Don't follow you,' said Filby.`

`'Can a cube that does not last for any time at all, have a real existence?'`

This document[4], contains six paragraphs representing direct speech. One of these paragraphs (paragraph 3) ends without the end-of-sentence punctuation, but the sentence `"All real things"` terminates at the end of that paragraph.

The following punctuation characters terminate sentences:

Character(s)	Name
`.`	Period
`!`	Exclamation point
`?`	Question mark

The following punctuation characters separate words but do not terminate sentences:

Character(s)	Name
`,`	Comma
`-`	Dash
`:`	Colon
`;`	Semicolon
`(`	Opening parenthesis
`)`	Closing parenthesis
`'`	Single quotation mark
`"`	Double quotation mark
`...`	Ellipses

The following punctuation characters may appear in the middle of a single word:

Character(s)	Name
`-`	Hyphen
`'`	Apostrophe

Note: A *dash* and a *hyphen* are represented by the same ASCII character in plain text. You can distinguish them using the following rule: a dash is always separated from other characters by spaces on both sides (e.g. `" woke - and left"`), while a hyphen is always preceded and followed by a letter (or a number) character (e.g., `"state-of-the-art"`).

Note: A *single quotation mark* and an *apostrophe* are represented by the same ASCII character in plain text. You can distinguish them by using the following rule: an apostrophe is always preceeded **and** followed by a letter character (e.g., `don't`, `can't`), while a single quotation mark must be either preceeded or succeeded by a space or another word/sentence-separating punctuation character (e.g. `this 'fun' thing` or `'This is it', he thought`).

[4] H.G. Wells, The Time Machine, http://www.gutenberg.org/cache/epub/35/pg35.html

This rule will cause apostrophes at the ends of the words (`boys'`, `Joneses'`) to be treated as single quotation marks, but this will not prevent proper recognition of the words, so we will let it slide.

Note: *Elipses* do not end the sentence. For a sentence to end, a fourth "dot" (a period) must be present (e.g., `So, this is where you are....`).

General Notes

You will write a number of function/method libraries for this assignment. It is up to each pair to decide which programming language to use. However, I do recommend that you select meaningful mainstream languages, as you may wind up wanting to reuse some of the code you developed in one (or more) of the subsequent labs, and, quite possibly, with a different partner. Students in a prior version of CSC 466 used Java, C, C++ and Python for the coursework (my recollection).

Work with CSV Files

We will interpret each line of a CSV file as a numeric vector. Different vectors may have different number of dimensions (depending on the input). You can represent missing numeric values as 0.

Your goal is to build a set of tools to work with CSV files and vectors of numbers that they contain. The tools must support the following functionality:

1. Opening a CSV file given its name/location on disk.
2. Reading individual vectors from the CSV file. Your code must correctly identify each value read from input, the end of the line (record/vector) and missing value situations.
3. Storing all vectors read from the input file in a main memory data structure (array, hash table, list, or any other data structure you choose). The data structure must be easily traversable (i.e., it should be relatively straightforward to traverse/loop over all vectors read from the CSV file).
4. Traversing the vectors retrieved from the CSV file.
5. Performing some computations on the retrieved vectors. In particular:
   - Computing the length of the vector (i.e, the distance from the point of origin).
   - Computing the dot product of a pair of vectors.
   - Computing the Eucledian distance between a pair of vectors.
   - Computing the Manhattan distance between a pair of vectors.
   - Computing the Pearson correlation between a pair of vectors.

- Computing the largest, the smallest, the mean, and the median value in the vector.
- Computing the largest, the smallest, the mean, and the median value in a single column for the entire collection of vectors.
- Computing the standard deviations of values within the vector and within a single column for the entire collection of vectors.

Note. There are special-purpose algorithms for median computation, but one of the most straightforward and practically usable ones is to sort the list numbers and then find the number in the middle. A more efficient method is described in this handout:

http://users.csc.calpoly.edu/~dekhtyar/349-Spring2010/lectures/lec13.349.pdf

Testing. You are tasked with providing me with evidence that the above-mentioned functionality has been implemented and works properly. Generally speaking, this means that you should write one or more programs that jointly use **all** the functionality described above. You can do it in a "unit test" manner (one small program per piece of functionality), or by writing a single program that tests everything, or by doing something in between.

Work with text files.

Your assignment for working with text files is similar. You will develop a library of methods/functions which jointly implement all the functionality listed below:

1. Opening a plain text file.
2. Reading the contents of the file, word-by-word, sentence-by-sentence, paragraph-by-paragraph.
3. Creating and maintaining a list of words found in the document.
4. Creating and maintaining a list of words found in the document with the frequency counts.
5. Counting the total number of words, different words, sentences, paragraphs found in the document.
6. Reporting the most frequent word/words, all words with frequencies equal to a given number, all words with frequencies exceeding a given number.
7. Checking if a word is found in a document.

Testing. Just as with CSV files, you are responsible for creating appropriate test programs that illustrate all the implemented functionality.

Data and Submission Instructions.

Data files. Some data files for both parts of the assignment are provided by the instructor (you can get them from the Lab 1 page on the home page of the course). You can also add your own data files for testing purposes.

Submission instructions. Submit all the source code you developed and all the extra data files you used. Include a README file which describes the contents of each submitted file (or, at least, each submitted program/library/class). Also include any compilation instructions in the README file. If you used `make`, submit the makefile, and include instructions in the README file. Include the names of the team members in each program and in the README file.

You will use the `handin` tool to submit your files. Each groups submits exactly one copy of all materials. To submit use the following command:

```
$ handin dekhtyar lab01-466 <files>
```

Demo. Each team will be asked to do a short demo during the lab period on Monday, September 27. The lab grade will not be assigned until after the demo.