# Privacy-Preserving Data Publishing
# Practical Work : Implementing Mondrian and the Laplace Mechanism

*You will have to send both a* ***PDF*** *report and your code commented to the email given in the syllabus before April the* $15^{th}$ *11 :59PM (Paris time).*
*The practical work session is written for programming in Java but you can program in another language among R, Python, Scala, provided (1) that the compiler/interpreter can be downloaded freely for Mac OS X and (2) that you give in a README file* ***all*** *the necessary and sufficient instructions for compiling/interpreting your code.*

---

**Goals**

1. Partition-based part :
   — Implement Mondrian for satisfying $k$-Anonymity
2. Differential privacy part :
   — Implement the Laplace mechanism for satisfying $\epsilon$-differential privacy
   — Analyze the errors due to the Laplace perturbation
   — Distribute the privacy budget

---

## 1 Partition-Based Models and Algorithms

In this section you are going to implement the Mondrian algorithm (Alg. 1), which can be used for producing equivalence classes satisfying $k$-Anonymity. You will test this algorithm over a dataset that you will have synthetically generated. The dataset consists in two `INT` columns and one `STRING` column. We assume that the first two columns are the quasi-identifier (`QID` below) and that the last one is a sensitive data (`SD` below).

1. Preliminary analysis :
   — Propose a data structure that represents an equivalence class
   — Propose a data structure that represents a set of tuples
2. Launch Eclipse and create a Java project
3. Write a class called `LaunchMe` which will serve as the entry point
   — `LaunchMe` inputs the following parameters :
      — The cardinality $n$ of the dataset
      — The min and max values of the two `QID` columns
      — The privacy parameter $k$
   — In `void main (String [])` :
      — Generate the tuples according to the given parameters (the `SD` value of each tuple will be chosen uniformly at random in a set of strings that you have to define) ;
      — Test the generation of your dataset. For printing it, you can for example :
         — Use the method `String Arrays.toString(int[])`
         — Print the tuples in a CSV file. Here is an example for writing `"column1, column2"` in a file :

```
BufferedWriter bw = new BufferedWriter(
  new FileWriter(
    new File ( "./myData.csv" ),
    false
  )
);
bw.append ( "column1, column2 \n" );
bw.flush();
bw.close();
```

4. Write a method called `mondrian` :
   — Input : a set of tuples (as defined previously) and the value of $k$
   — Outputs : the equivalence classes computed according to the Mondrian algorithm
5. Print in a CSV file the dataset generated, the $k$-Anonymity level, and the resulting equivalence classes.

---

**Algorithm 1:** `MondrianAnonymize`

**input** : A partition $\mathcal{P}$ to split
**output**: A set of partitions, each containing between $k$ and $2k-1$ tuples

**if** *no allowable multidimensional cut for partition* **then return** $\mathcal{P}$;
**else**
   $dim \leftarrow$ `chooseDimension()`;
   $fs \leftarrow$ `frequencySet(`$\mathcal{P}$, $dim$`)`;
   $splitVal \leftarrow$ `findMedian(`$fs$`)`;
   $\mathcal{L} \leftarrow \{t \in \mathcal{P} : t.dim \leq splitVal\ \}$;
   $\mathcal{R} \leftarrow \{t \in \mathcal{P} : t.dim > splitVal\ \}$;
   `MondrianAnonymize(`$\mathcal{L}$`)`;
   `MondrianAnonymize(`$\mathcal{R}$`)`;

---

## 2 Differential Privacy

In this section you are going to implement the Laplace Mechanism for satisfying $\epsilon$-differential privacy. You will test your implementation on a set of integers randomly generated, where each integer represents the data of a distinct individual (*e.g.,* a salary).

1. Launch Eclipse and create a Java project
2. Write a class `LaunchMe` that will serve as the entry point
   — Inputs :
      — The size $n$ of the set of integers
      — The max value $m$ of the integers (we fix the min value to 0)
      — The value of the privacy parameter $\epsilon$
   — In `void main (String [])` :
      — Generate the set of integers (you can use `double Math.random()` for generating a float uniformaly at random in `[0, 1[` and on `int Math.round(float)` in order to get the closest integer)
      — Test on small sizes
3. Write a class called `Laplace` in charge of generating the Laplace perturbation such that :
   — $\epsilon$ is a parameter of the `Laplace` object constructor
   — The method `double Laplace.genNoise(int, double)` generates a random variable that follows the Laplace distribution and that is called before perturbing an aggregate query :
      — The first parameter (type `int`) is the sensitivity of the aggregate query to perturb

  — The second parameter (type `double`) is a float in $]0, 1]$ representing the fraction of $\epsilon$ to consume for this call
  — In order to generate a random variable that follows a Laplace distribution, you can (1) use `double Math.random()` for having a uniform random variable, and (2) apply the mathematical transform from a uniform variable to a Laplace variable described here : `https://en.wikipedia.org/wiki/Laplace_distribution#Generating_random_variables_according_to_the_Laplace_distribution`.
— Note that :
  — A `Laplace` object stops returning any noise as soon as its $\epsilon$ is entirely consumed
  — A `Laplace` object must have a `TEST` mode, which, when enabled, considers that the privacy budget is infinite (no budget consumption by any query)

4. **Test** `genNoise` as follows :
— Enable the `TEST` mode
— Choose an $\epsilon$ value and a sensitivity value
— Generate a large number of perturbations (*e.g.,* $10^4$ should be enough)
— Count the number of perturbations that fall in the range $]-500, -480]$, those that fall in $]480, 460]$, $\ldots$, and those that fall in $]480, 500]$ (the histogram of the perturbations generated)
— Print the ranges and their corresponding counts in a CSV file, open the file with *e.g.* Open Office, and plot ranges and counts in a graph
— Compare your graph to the Laplace distribution that you should obtain (where $b$ is sensitivity/$\epsilon$) : `http://keisan.casio.com/exec/system/1180573177`

5. **Why must we limit the number of aggregates published ?**
Enable the `TEST` mode. Let $n = 10^3$ and $\epsilon = 10^-4$ :
   (a) Formulate a `COUNT` on your set of integers (*e.g.,* count the number of integers greater than 10)
   (b) Compute the true `COUNT` value : $r$
   (c) Generate 10 perturbations from $p_1$ to $p_{10}$, obtain $r'_1 = r + p_1$ to $r'_{10} = r + p_{10}$ by perturbing $r$ 10 times, and compute $a_{10}$ the average of the $r_i$.
   (d) Do the same with $10^2$ perturbations, $10^3$ perturbations, $10^4$ perturbations, $10^5$ perturbations, and $10^6$ perturbations.
   (e) Plot in a graph (*e.g.,* Open Office file) : on the x-axis the number of perturbations, and on the y-axis the averages. Plot also the true count
   (f) How many perturbations are needed for being *close* to the true count (*e.g.,* $\pm 10\%$ difference)

6. **How big is the error due to the perturbation ?**
Lets the *error* be the absolute value of the perturbation. Enable the `TEST` mode. Let $n = 10^3$, $\epsilon = 10^-2$, and $m = 1000$, :
— Generate $10^3$ perturbations for a `SUM` aggregate, compute the error due to each perturbation, and compute the average error $err_{avg}$
— Does $err_{avg}$ depend on the dataset size ? On the dataset values ?
— What is the ratio between $err_{avg}$ and the Laplace scale factor parameter (*i.e.,* sensitivity/$\epsilon$) ?
— With varying dataset sizes, *i.e.*, $n \in \{10^2, 10^3, 10^4, 10^5, 10^6\}$ :
   (a) Compute the true `SUM` of the dataset values : $sum$
   (b) Compute the ratio between the previously computed average error and $sum$ : $err_{avg}/sum$
   (c) Print $n$ together with its corresponding ratio in a CSV file
— Plot in a graph the evolution of the ratio with respect to the various dataset sizes $n$.
— How many tuples are « needed » for making this ratio small enough ? (*e.g.,* 10%)