# Taller 5: El algoritmo KMeans [*]


Juan Francisco Díaz Frias  Emily Núñez

Octubre 2024

## 1. Ejercicios de programación

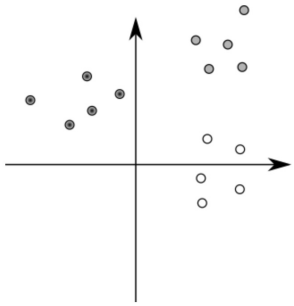
En este taller se implementará el algoritmo *KMeans* para detectar clusters. Este algoritmo se usa para hacer una partición de un conjunto de $n$ vectores en $k$ clusters. Aquí, los vectores se separan en grupos en función de su similitud: es más probable que los vectores que están más cerca entre sí en el espacio terminen en el mismo grupo, y es probable que los vectores distantes estén en diferentes grupos. El algoritmo *KMeans* tiene muchas aplicaciones, por ejemplo en minería de datos, filtros de imágenes y procesamiento de señales.

El objetivo de este taller es verificar que el uso de la concurrencia (de tareas y/o de datos) efectivamente logra mejoras en el tiempo de ejecución de los programas que las usan.

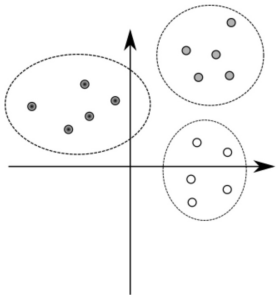
### Preliminares

A continuación un ejemplo sencillo para ilustrar la idea del algoritmo. Suponga que se tiene un conjunto de vectores en el espacio 2D, como se muestra en la figura:

[*] Ejercicio adaptado de los cursos de coursera de la EPFL sobre programación funcional y programación concurrente



Un humano, puede distinguir con solo mirar la gráfica, tres *clusters* (agrupaciones):



Cuando la dimensión, el número de vectores y el número de *clusters* crecen, se vuelve muy difícil e incluso imposible determinar manualmente los *clusters*. El algoritmo *KMeans* es un algoritmo sencillo que recibe un conjunto de vectores (llamados puntos) y devuelve un conjunto de *clusters* de la siguiente manera:

1. Escoge $k$ puntos llamados medianas. Esta fase se llama **inicialización**.

2. Asocia cada punto del conjunto de entrada con la mediana más cercana a él, obteniendo así $k$ *clusters* de puntos. Esta es la fase de **clasificación** de los puntos.

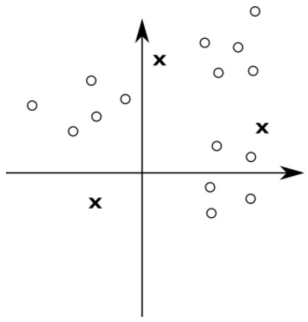
3. Actualiza cada mediana con el valor promedio de los puntos del *cluster* correspondiente. Esta es la fase de **actualización**.

4. Si las $k$ nuevas medianas han cambiado de forma significativa, se repite la fase 2 de clasificación. Sino, se dice que el algoritmo **converge**. Las $k$ medianas calculadas representan los diferentes *clusters*, donde cada punto pertenece al *cluster* correspondiente a la mediana más cercana.

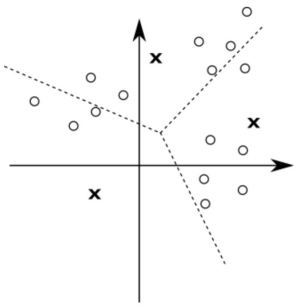
En este punto, dos de las etapas necesitan discusión adicional. En primer lugar, ¿Cómo escoger las $k$ medianas iniciales? La fase de inicialización se puede hacer de diferentes maneras. En este taller se tomarán al azar del conjunto de puntos de entrada. En segundo lugar, ¿cómo se determina que el algoritmo converge? En este taller se verificará para

cada mediana, que la distancia al cuadrado entre ella y la nueva mediana sea inferior a un valor predefinido **eta**.

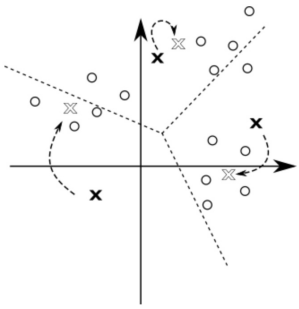
Para ilustrar mejor esta idea, se ilustrará gráficamente la aplicación del algoritmo *kmeans*. En primer lugar, se escogen al azar $k$ medianas iniciales, mostradas con una $\times$ en la figura:



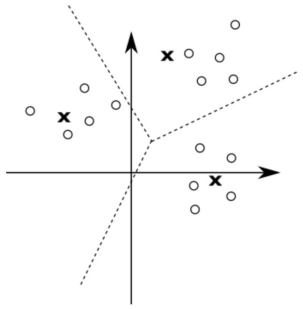
Luego, se clasifican los puntos de acuerdo a la mediana más cercana. Las medianas dividen el espacio en regiones, donde cada punto de un cluster está más cerca a la mediana representante del cluster que a cualquier otra mediana, como se muestra en la figura siguiente:



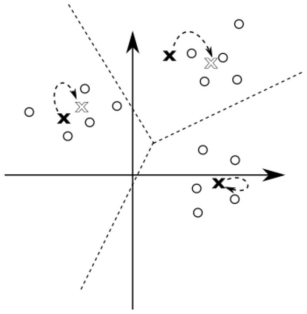
Todos los puntos en la misma región forman un cluster. Una vez se han clasificado los puntos, se actualizan los valores de cada mediana al promedio de los puntos del cluster:



En este caso cada mediana fue actualizada de forma significativa. Esto indica que el algoritmo todavía no converge. Por tanto, se repiten las fases nuevamente. Primero, clasificar los puntos:



Y luego, actualizar las medianas de nuevo:



Una de las medianas no cambió para nada en la última fase. Sin embargo, las otras medianas han cambiado de forma significativa, por lo que se continúa el proceso hasta que el cambio en la posición de cada mediana sea inferior al valor **eta**.

En cada iteración del algoritmo *kmeans*, la asociación de puntos a los clusters y el cálculo del promedio de los clusters se pueden hacer en paralelo. Nótese que la asociación de un punto a su cluster es independiente de los otros puntos en la entrada y que, de forma similar, el cálculo del porcentaje de un cluster es independiente de los otros clusters. Y una vez todas estas tareas en paralelo se hayan completado, el algoritmo puede seguir con la siguiente iteración.

El algoritmo *kmeans* es un ejemplo de algoritmo paralelo síncrono masivo (*bulk synchronous parallel algorithm*). Este tipo de algoritmos se compone de una secuencia de fases, cada una de las cuales contiene:

- computación en paralelo, en la cual varios procesos se desarrollan independientemente en computaciones locales y producen algunos valores;
- comunicación, en la cual los procesos intercambian datos;

- y, sincronización de carrera, durante la cual los procesos esperan hasta que otros procesos terminen.

Los modelos de programación de datos paralelos suelen ser una buena opción para los algoritmos paralelos síncronos masivos ya que cada fase síncrona masiva puede corresponder a una cierta cantidad de operaciones de datos paralelos.

Su tarea en este taller consistirá en implementar cada una de las fases del algoritmo, tanto en su versión secuencial como en su versión paralela, y analizar la tasa de aceleración sobre las soluciones secuenciales cuando se usan las soluciones concurrentes. Por eso cada fase del algoritmo tendrá dos versiones: una secuencial y otra concurrente. Sin embargo, ambas versiones tendrán la misma especificación. Esto significa que una vez usted haya implementado la versión secuencial, su versión concurrente debe ser el resultado de usar paralelismo de tareas y/o paralelismo de datos sobre la versión secuencial.

## 1.1. Clasificando los puntos

En esta primera parte del taller, se implementará la fase de clasificación de los puntos de acuerdo al cuadrado de la distancia a las medianas, como se describió anteriormente.

Para ello se usará la clase *Punto* para representar los vectores de entrada al algoritmo:

```
class Punto(val x: Double, val y: Double) {
  private def cuadrado(v: Double): Double = v * v

  def distanciaAlCuadrado(that: Punto): Double =
    cuadrado(that.x - x) + cuadrado(that.y - y)

  private def round(v: Double): Double = (v * 100).toInt / 100.0

  override def toString = s"(round(x),{round(y)})"
}
```

Implemente la función de clasificación en sus versiones secuencial y concurrente, `clasificarSeq` y `clasificarPar`, como se indica a continuación:

```
  def clasificarSeq(puntos: Seq[Punto], medianas: Seq[Punto]): Map[Punto, Seq[Punto]] = {
     ???
  }

def clasificarPar(umb:Int)(puntos: Seq[Punto], medianas: Seq[Punto]): Map[Punto, Seq[Punto]]   = {
      ???
  }
```

[**Ayuda 1**] Utilice el método `groupBy` sobre colecciones, y la función `hallarPuntoMasCercano` cuyo código se presenta a continuación:

```
  def hallarPuntoMasCercano(p: Punto, medianas: Seq[Punto]): Punto ={
    assert(medianas.nonEmpty)
    medianas.map(pto=>(pto,p.distanciaAlCuadrado(pto))).sortWith((a,b)=>(a._2<b._2)).head._1
  }
```

[**Ayuda 2**] La función `clasificarPar` debe ser una versión usando paralelismo de tareas. Por ello recibe un parámetro adicional, que es el umbral bajo el cual se invoca a la función secuencial. Y si el número de puntos está por encima del umbral, se hace uso del paralelismo de tareas.

## 1.2. Actualizando las medianas

En la segunda parte de este taller, se implementará la fase de actualización de las medianas correspondientes a cada cluster. Implemente la función de actualización en sus versiones secuencial y concurrente, `actualizarSeq` y `actualizarPar`, como se indica a continuación:

```
def actualizarSeq(clasif: Map[Punto, Seq[Punto]], medianasViejas: Seq[Punto]): Seq[Punto] = {
    ???
}

def actualizarPar(clasif: Map[Punto, Seq[Punto]], medianasViejas: Seq[Punto]): Seq[Punto] = {
   ???
}
```

Estas funciones, reciben la asociación (map) de los puntos clasificados en la fase anterior , y las medianas vigentes, y devuelven la nueva colección de medianas actualizadas según lo mencionado en la descripción de esa etapa.

Tenga cuidado de preservar el orden en la colección resultante: la mediana $i$ en la colección nueva, debe corresponder a la mediana $i$ en la colección vieja.

[**Ayuda**] Utilice las funciones `calculePromedioSeq` y `calculePromedioPar` cuyo código se presenta a continuación:

```
def calculePromedioSeq(medianaVieja: Punto, puntos: Seq[Punto]): Punto = {
  if (puntos.isEmpty)  medianaVieja
  else {

    new Punto(puntos.map(p=>p.x).sum / puntos.length, puntos.map(p=>p.y).sum / puntos.length)
  }
}


def calculePromedioPar(medianaVieja: Punto, puntos: Seq[Punto]): Punto = {
  if (puntos.isEmpty)  medianaVieja
  else {
    val puntosPar = puntos.par
    new Punto(puntosPar.map(p=>p.x).sum / puntos.length, puntosPar.map(p=>p.y).sum / puntos.length)
  }
}
```

## 1.3. Detectando convergencia

Finalmente, se implementará el criterio de detección de convergencia del algoritmo en las dos versiones: secuencial y paralela. El algoritmo converge si dadas dos colecciones de secuencias, las viejas y las nuevas, la distancia entre cada mediana nueva y su correspondiente mediana antigua, es inferior a un valor *eta* dado. Las funciones que detectan la convergencia reciben entonces el parámetro *eta* y las dos colecciones de medianas (la antigua y la nueva) y devuelven un booleano indicando si hay convergencia o no:

```
def hayConvergenciaSeq(eta: Double, medianasViejas: Seq[Punto],
                       medianasNuevas: Seq[Punto]): Boolean ={
   ???
}

def hayConvergenciaPar(eta: Double, medianasViejas: Seq[Punto],
                          medianasNuevas: Seq[Punto]): Boolean ={
     ???
}
```

Implemente las funciones `hayConvergenciaSeq` y `hayConvergenciaPar`.
La función `hayConvergenciaSeq` debe ser iterativa.
La función `hayConvergenciaPar` debe utilizar paralelismo de tareas.

## 1.4. Implementando el algoritmo *kmeans*

Ahora se tiene todo listo para implementar el algoritmo *kmeans*. Sólo hace falta combinar las funciones implementadas de la forma correcta.

Implemente las funciones `kMedianasSeq` y `kMedianasPar` indicadas a continuación, de forma iterativa, es decir, deben ser recurrentes por la cola (por eso la anotación @tailrec):

```
@tailrec
final def kMedianasSeq(puntos: Seq[Punto], medianas: Seq[Punto], eta: Double): Seq[Punto] ={
  ...
}

@tailrec
final def kMedianasPar(puntos: Seq[Punto], medianas: Seq[Punto], eta: Double): Seq[Punto] ={
  ...
}
```

Nótese que estas funciones reciben una colección de puntos, y una colección de medianas iniciales, y devuelven la colección de medianas correspondientes a cada cluster, es decir las medianas correspondientes a aplicar el algoritmo *kmeans* en su versión correspondiente, correctamente, es decir hasta que se cumpla el criterio de convergencia.

## 1.5. Corriendo el algoritmo

Para correr los algoritmos, y probarlos, y analizar el efecto de usar las versiones paralelas o no, incluya en su paquete las siguientes funciones:

```
def generarPuntos(k: Int, num: Int): Seq[Punto] ={
  val randx = new Random
  val randy = new Random
  (0 until num)
    .map({ i =>
      val x = ((i + 1) %k) * 1.0 / k + randx.nextDouble() * 0.5
      val y = ((i + 5) %k) * 1.0 / k + randy.nextDouble() * 0.5
      new Punto(x, y)
    })
}

def inicializarMedianas(k: Int, puntos: Seq[Punto]): Seq[Punto] ={
  val rand = new Random
  (0 until k).map(_ => puntos(rand.nextInt(puntos.length)))
}
```

La función `generarPuntos` recibe el número de *clusters* esperado $k$ y el número de puntos a generar *num*, y devuelve, *num* puntos generados al azar. Por su parte, la función `inicializarMedianas` recibe el número de *clusters* esperado, $k$ y los puntos generados, *puntos* y devuelve $k$ puntos escogidos entre los *puntos* recibidos, los cuales fungirán como las primeras $k$ medianas del proceso.

Luego invóquelas desde su *worksheet* de pruebas muchas veces, variando el número de puntos sobre los cuales se aplica el algoritmo (cientos, miles, cientos de miles, millones),

el número de clusters a calcular (2, 4, 8, 16, 32, 64, 128 y 256) y el parámetro de convergencia (0.01, 0.001). Haga una tabla con los datos recolectados y analice los resultados en términos de aceleración.

Para facilitar la recolección de estos datos, en el paquete *Benchmark* asociado a este taller, estamos proveyendo las funciones denominadas `tiemposKmedianas` y `probarKmedianas`.

La primera, `tiemposKmedianas`, sirve para recolectar tiempos de las soluciones secuencial y paralela y la aceleración, en una entrada particular. Esta función recibe una secuencia de puntos a clasificar (generada con `generarPuntos`), el número de clusters con la que se generó, y un valor *eta* de precisión para terminar, y devuelve una tripleta con los tiempos tomados por los algoritmos secuencial y paralelo para resolver esa entrada, y la aceleración correspondiente.

Por ejemplo:

```
import Benchmark._
import kmedianas2D._

val puntos16_3 = generarPuntos(3, 16).toSeq
tiemposKmedianas(puntos16_3, 3, 0.01)
```

da como resultado:

```
import Benchmark._
import kmedianas2D._

val puntos16_3: Seq[kmedianas2D.Punto] = Vector((0.46, 1.0), (0.94, 0.12), (0.43, 0.46), (0.47, 0.88),
    (0.8, 0.34), (0.24, 0.39), (0.57, 1.15), (1.15, 0.2), (0.15, 0.52), (0.58, 1.12), (0.96, 0.43),
    (0.16, 0.79), (0.44, 1.11), (0.85, 0.11), (0.27, 0.5), (0.49, 0.83))
val res4: (org.scalameter.Quantity[Double], org.scalameter.Quantity[Double], Double) =
    (0.021625 ms,0.832845 ms,0.02596521561635118)
```

Como pueden ver la versión paralela consumió mucho más tiempo que la secuencial (el número de puntos era muy pequeño para que valiera la pena paralelizar).

Pero si la prueba que hacemos es la siguiente:

```
val puntos32768_256 = generarPuntos(256, 32768).toSeq
tiemposKmedianas(puntos32768_256,256,0.01)
```

pueden ver que el resultado es:

```
val puntos32768_256 = ...
val res5: (org.scalameter.Quantity[Double], org.scalameter.Quantity[Double], Double) =
(2102.119734 ms,409.530812 ms,5.132995301950564)
```

o sea, hay una aceleración mayor a 5 veces por usar la versión paralela.

Obviamente, antes de analizar los tiempos, es necesario estar seguros que sus algoritmos están entregando las respuestas correctas. Para ello, además de asegurarse que sus argumentaciones de corrección están bien hechas, vamos a usar la función `probarKmedianas`, la cual utiliza la librería *Plotly* para desplegar los resultados de sus algoritmos gráficamente, trabajando en 2 dimensiones, al mismo tiempo que devuelve los tiempos de los algoritmos en secuencial y paralelo y su aceleración. O sea, la misma tripleta que devolvía la anterior función.

Para poder usar la librería *Plotly* incluya la siguiente línea en el archivo *build.sbt*:

```
libraryDependencies += "org.plotly-scala" %% "plotly-render" % "0.8.1"
```

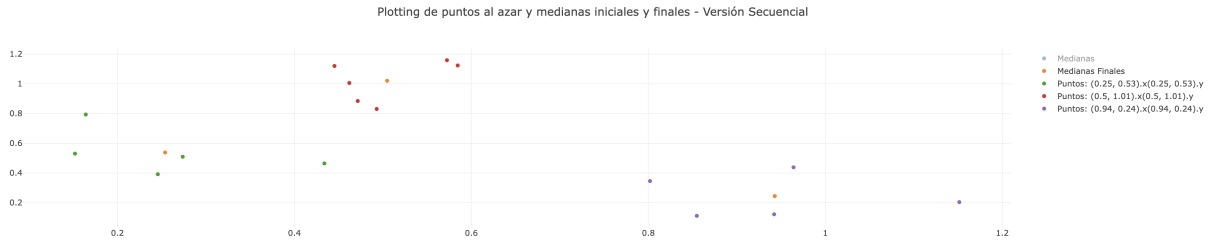
La función, `probarKmedianas` recibe al igual que `tiemposKmedianas` los puntos a clasificar, el número de *clusters* y el parámetro *eta*, pero además de devolver los tiempos tomados por los algoritmos secuencial y paralelo para hacer la clasificación, y su aceleración en una tripleta, genera unos archivos html, que se pueden observar con cualquier navegador, y que le permiten ver gráficamente la tarea de clasificación de ambos algoritmos (el secuencial y el paralelo).

Por ejemplo:

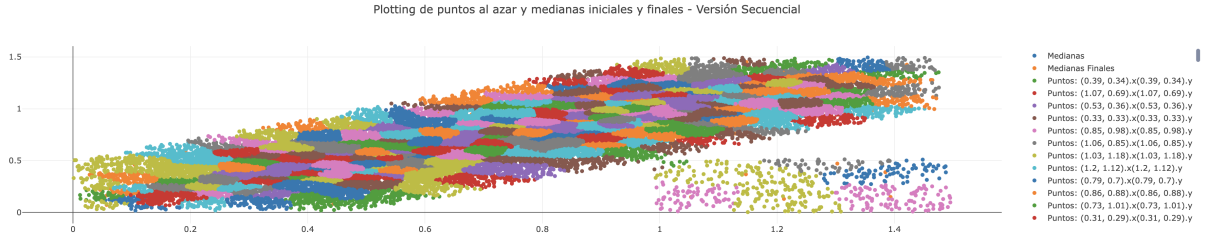
```
import Benchmark._
import kmedianas2D._

val puntos16_3 = generarPuntos(3, 16).toSeq
probarKmedianas(puntos16_3, 3, 0.01)
```

además de devolver una tripleta, muy similar a la que devolvió `tiemposKmedianas` (es el mismo ejemplo), genera dos archivos html (uno para la versión secuencial y otro para la versión paralela) con el resultado de la clasificación, coloreando cada cluster con un color, como se muestra en la figura siguiente para la versión secuencial.



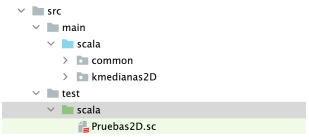

Para el caso de miles de puntos, la imagen se puede ver como la siguiente:




# 2. Entrega

## 2.1. Paquete *kmedianas* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:




Las funciones correspondientes a cada ejercicio, `clasificarSeq`, `clasificarPar`, `actualizarSeq`, `actualizarSeq`, `hayConvergenciaSeq`, `hayConvergenciaPar`, `kMedianasSeq` y `kMedianasPar` deben ser implementadas en un paquete de Scala denominado `kmedianas2D`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object kmedianas2D {
package object kmedianas2D {
  import scala.annotation.tailrec
  import scala.collection.{Map, Seq}
  import scala.collection.parallel.CollectionConverters._
  import scala.util.Random
  import common._

  // Definiciones comunes para las dos versiones (secuencial y paralela)

  class Punto(val x: Double, val y: Double) {
    private def cuadrado(v: Double): Double = v * v

    def distanciaAlCuadrado(that: Punto): Double =
      cuadrado(that.x - x) + cuadrado(that.y - y)

    private def round(v: Double): Double = (v * 100).toInt / 100.0

    override def toString = s"(${round(x)},${round(y)})"
  }

  def generarPuntos(k: Int, num: Int): Seq[Punto] ={
    val randx = new Random(1)
    val randy = new Random(3)
    (0 until num)
      .map({ i =>
        val x = ((i + 1) % k) * 1.0 / k + randx.nextDouble() * 0.5
        val y = ((i + 5) % k) * 1.0 / k + randy.nextDouble() * 0.5
        new Punto(x, y)
      })
  }

  def inicializarMedianas(k: Int, puntos: Seq[Punto]): Seq[Punto] ={
    val rand = new Random(7)
    (0 until k).map(_ => puntos(rand.nextInt(puntos.length)))
  }


  // Clasificar puntos
  def hallarPuntoMasCercano(p: Punto, medianas: Seq[Punto]): Punto ={
    assert(medianas.nonEmpty)
    medianas.map(pto=>(pto,p.distanciaAlCuadrado(pto))).sortWith((a,b)=>(a._2<b._2)).head._1
  }

  // Versiones secuenciales

  def calculePromedioSeq(medianaVieja: Punto, puntos: Seq[Punto]): Punto = {
    if (puntos.isEmpty)  medianaVieja
    else {

      new Punto(puntos.map(p=>p.x).sum / puntos.length, puntos.map(p=>p.y).sum / puntos.length)
    }
  }

  def clasificarSeq(puntos: Seq[Punto], medianas: Seq[Punto]): Map[Punto, Seq[Punto]] = {
    ...
  }

  def actualizarSeq(clasif: Map[Punto, Seq[Punto]], medianasViejas: Seq[Punto]): Seq[Punto] = {
    ...
  }

  @tailrec
  def hayConvergenciaSeq(eta: Double, medianasViejas: Seq[Punto], medianasNuevas: Seq[Punto]): Boolean ={
...
```

```
  }

  @tailrec
  final def kMedianasSeq(puntos: Seq[Punto], medianas: Seq[Punto], eta: Double): Seq[Punto] ={
...
  }

  // Versiones paralelas
  def calculePromedioPar(medianaVieja: Punto, puntos: Seq[Punto]): Punto = {
    if (puntos.isEmpty)  medianaVieja
    else {
      val puntosPar = puntos.par
      new Punto(puntosPar.map(p=>p.x).sum / puntos.length, puntosPar.map(p=>p.y).sum / puntos.length)
    }
  }

  def clasificarPar(umb:Int)(puntos: Seq[Punto], medianas: Seq[Punto]): Map[Punto, Seq[Punto]] = {
...
  }

  def actualizarPar(clasif: Map[Punto, Seq[Punto]], medianasViejas: Seq[Punto]): Seq[Punto] = {
...
  }

  def hayConvergenciaPar(eta: Double, medianasViejas: Seq[Punto], medianasNuevas: Seq[Punto]): Boolean ={
...
  }

  @tailrec
  final def kMedianasPar(puntos: Seq[Punto], medianas: Seq[Punto], eta: Double): Seq[Punto] ={
...
  }
}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import kmedianas2D._
import org.scalameter._
import plotly._, element._, layout._

def tiempoDe[T](body: => T) = {
  val timeA1 = config(
    KeyValue(Key.exec.minWarmupRuns -> 20),
    KeyValue(Key.exec.maxWarmupRuns -> 60),
    KeyValue(Key.verbose -> false)
  ) withWarmer(new Warmer.Default) measure (body)
  timeA1
}

def tiemposKmedianas(numPuntos:Int, k:Int, eta:Double) = {
...
}

def probarKmedianas(numPuntos:Int, eta:Double, k:Int) = {
...
}
// Pruebas desarrolladas...
...
...
...
```

## 2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de corrección de las funciones implementadas, informe de desempeño de las funciones secuenciales y de las funciones paralelas, y análisis comparativo de las diferentes soluciones. El

primero, como se ha venido haciendo en los otros talleres, corresponde a una argumentación del por qué están bien implementadas las funciones secuenciales solicitadas, siempre que las versiones paralelas sean, algorítmicamente hablando, las mismas . El segundo y el tercero, se pueden hacer usando las funciones `generarPuntos` e `inicializarMedianas` provistas en el paquete *kmedianas2D* y `tiemposKmedianas` y `probarKmedianas` provistas en el paquete *Benchmark*.

### 2.2.1. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de las funciones entregadas.

Para cada función secuencial del proceso de cálculo de las $k$ medianas, , `clasificarSeq, actualizarSeq, hayConvergenciaSeq` y `kMedianasSeq` argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

Además para cada función paralela correspondiente, `clasificarPar, actualizarPar, hayConvergenciaPar` y `kMedianasPar` argumente que son algorítmicamente las mismas versiones seuenciales. Y utilice las gráficas generadas con la función `probarKmedianas` del paquete *Benchmark* para analizar la corrección de los resultados entregados por sus funciones.

### 2.2.2. Informe de desempeño de las funciones secuenciales y de las funciones paralelas

Su informe debe presentar una tabla con los resultados en tiempos y aceleración de correr las versiones secuencial y concurrente, `kMedianasSeq` y `kMedianasPar` . Es importante que genere diversos ejemplos con secuencias de puntos del miso tamaño, variando el número de *clusters* y hacerlo para muchos tamaños (se sugiere que las dimensiones de la secuencia de puntos y del número de *clusters* sean potencias de 2, obviamente mucho más grandes las de los puntos que las de los *clusters*). No olvide describir cómo generó los ejemplos de pruebas.

### 2.2.3. Análisis comparativo de las diferentes soluciones

Su informe debe presentar diferentes análisis basado en los resultados.

Para el caso de paralelismo de tareas, analizar cada versión secuencial versus su versión paralela correspondiente. Para el caso de paralelismo de datos, indicar claramente dónde se usó. ¿Las paralelizaciones sirvieron? ¿Cuándo es mejor usar paralelismo de tareas? ¿Cuándo es mejor usar paralelismo de datos? ¿Se pueden combinar las dos?

### 2.3. Fecha y medio de entrega

Todo lo anterior, es decir los archivos `package.scala`, `pruebas.sc`, e `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 31 de octubre de 2024, en un archivo comprimido que traiga estos tres elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación concurrente*.

## 3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.
- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.