

# K-Means opdracht

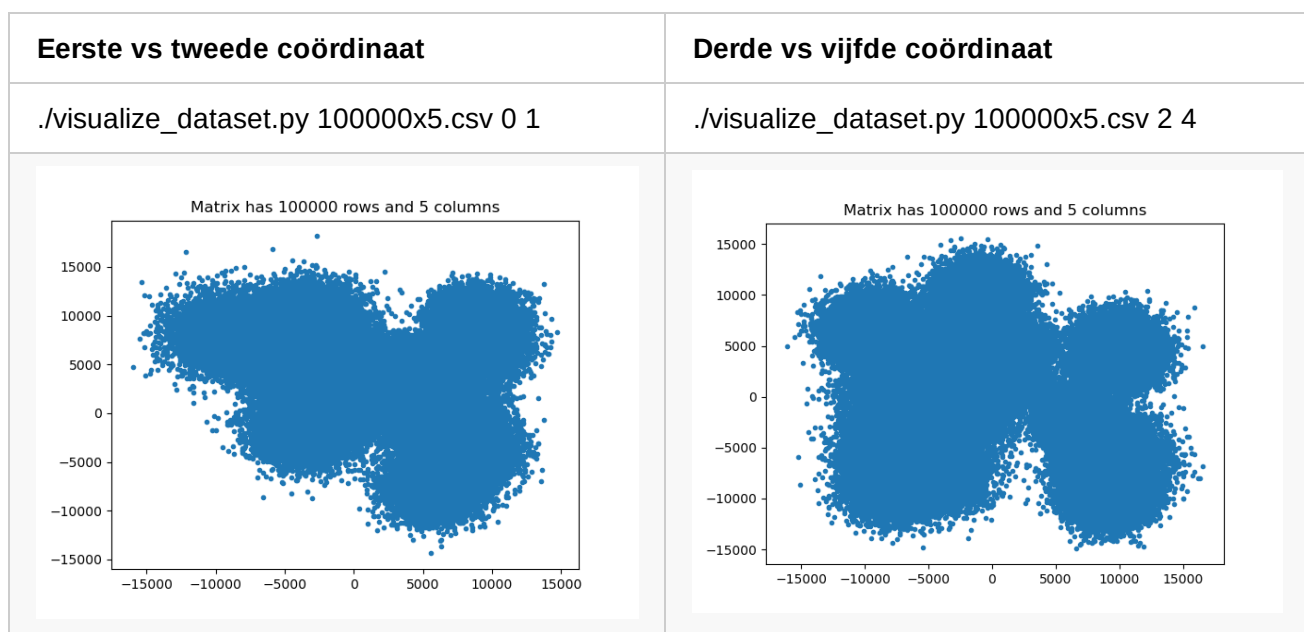
Het k-means algoritme is een **machine learning** techniek die in de categorie van '**unsupervised learning**' valt. Gegeven een (typisch grote) **hoop data**, probeert het algoritme deze data in een aantal **groepen** te verdelen, die men **clusters** noemt. Het aantal zulke clusters wordt gewoonlijk **k** genoemd, vandaar ook de 'k' in k-means.

## Input data

De data bestaat uit een aantal N-dimensionele punten; elk punt wordt dus voorgesteld door N reële getallen. Hieronder vind je een voorbeeld voor de `100000x5.csv` dataset, die **100000 punten bevat, elk 5-dimensioneel**:

```
4434.235023,5907.544715,7032.841276,667.7167487,5184.414441
9302.218828,7872.245686,7405.934857,-11086.42565,-33.24185058
5572.140527,3635.807919,8640.109276,-2089.622984,4258.818933
-8539.769531,5622.614758,718.1220525,2519.842204,-1569.796239
6577.706423,1684.171215,-8512.036134,4872.428015,-7276.385818
... (nog 99995 punten) ...
```

Met het Python script `visualize_dataset.py`, dat verderop nog toegelicht wordt, kan je zo'n dataset visualiseren. Omdat we niet zomaar 5-dimensionele gegevens kunnen tonen, kiezen we twee van de vijf dimensies die we willen tonen op een 2D plot: in het voorbeeld links wordt de eerste coördinaat (index 0) gebruikt op de x-as van de plot en de tweede coördinaat (index 1) op de y-as. Het voorbeeld aan de rechterkant kiest de derde coördinaat voor de x-as, en de vijfde voor de y-as.



# Het algoritme

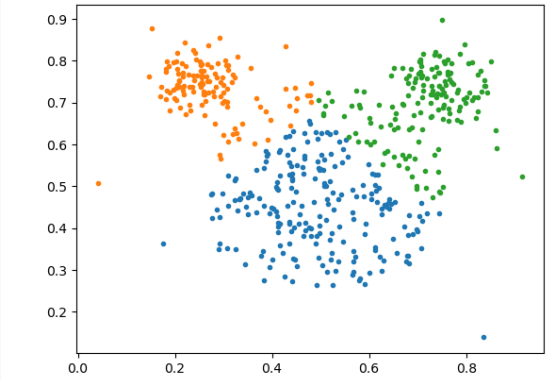
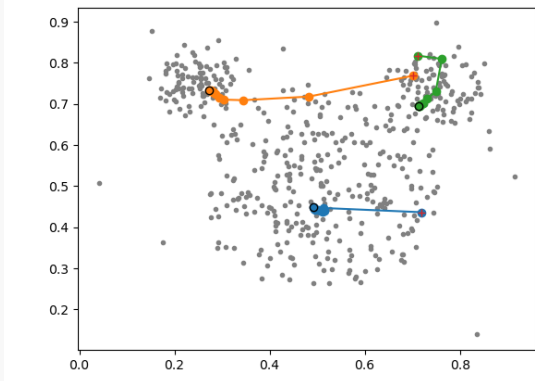
Uiteindelijk zal het k-means algoritme aan elk van de punten een label toekennen dat zegt tot welke cluster het hoort: wat typisch gedaan wordt is voor een punt een getal tussen 0 en k-1 opgeven om zo weer te geven tot welk van de k clusters het hoort.

Het algoritme vindt zo'n clustering door op zoek te gaan naar k centroids, de centrale punten (ook N-dimensioneel) van de clusters. Deze centroids leggen vast tot welke cluster een datapunt hoort, dit is namelijk de dichtstbijzijnde centroid wat het kwadraat van de euclidische afstand betreft:

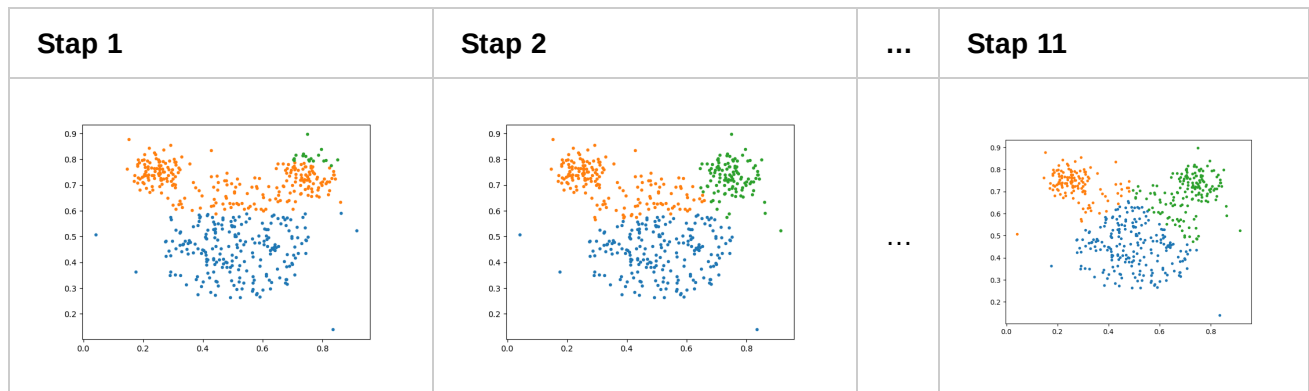
$$\text{afstand tussen punt } P \text{ en centroid } C = \sum_{i=0}^{N-1} (P_i - C_i)^2$$

De centroids zelf hoeven uiteraard geen punten uit de dataset te zijn. Het algoritme start van willekeurige centroids, berekent voor elk punt bij welke cluster het hoort (welke de dichtstbijzijnde centroid is), en berekent dan nieuwe centroids. De nieuwe waarde van centroid j is het gemiddelde van de punten die tot die cluster j horen (vandaar de 'means' in k-means). Deze procedure wordt herhaald totdat de toegekende clustering niet meer verandert.

De onderstaande figuren tonen enkele van deze zaken voor de mouse\_500x2.csv dataset (hierin zijn de punten 2D, wat het plotten makkelijk maakt), wanneer er naar 3 clusters gezocht wordt. Aan de linkerkant wordt de finale clustering getoond: elk punt heeft een kleur gekregen volgens de cluster waartoe het hoort. Aan de rechterkant kan je zien hoe de centroids aangepast worden: de initiële centroids worden met een '+' aangeduid, en zijn willekeurig gekozen. In dit voorbeeld zijn er 11 stappen nodig om tot de finale centroids, aangegeven met een 'o', te komen, en het zijn deze die de clusters aan de linkerzijde vastleggen.

Gevonden clusters (k=3)	Evolutie centroids (debug optie)
<code>./visualize_clusters.py mouse_500x2.csv output.csv 0 1</code>	<code>./visualize_centroids.py mouse_500x2.csv centroidtrace.csv 3 0 1</code>
	

Voor elk van deze 11 stappen kan je ook de bijhorende clustering tonen, zoals in de onderstaande figuren. De figuur bij 'stap 1' toont de clustering die hoort bij de willekeurige initiële centroids, terwijl de laatste stap terug de finale clustering is.



De clusters die op basis van de beschreven procedure gevonden worden zullen in het algemeen afhangen van de keuze van de initiële centroids, en de procedure herhalen kan dus ook tot een betere clustering (in de zin dat de som van de afstanden tot de centroids kleiner is) leiden. We zullen daarom de bovenstaande procedure een aantal keren, **repetitions** genoemd, herhalen, en de beste gevonden clustering bewaren.

In pseudo-code ziet de gehele procedure er als volgt uit:

```

bestClusters = None
bestDistanceSquaredSum = Infinity

for r in range(repetitions):

    centroids = choose_centroids_at_random(k) # (use Rng to pick k random points)
    clusters = [ -1, ..., -1 ] # initially we don't know the closest centroid index
                                # for each point

    changed = True
    while changed:

        changed = False
        distanceSquaredSum = 0

        for p in range(numberOfPoints)
            newCluster, dist = find_closest_centroid_index_and_distance(p, centroids)
            distanceSquaredSum += dist

            if newCluster != clusters[p]:
                clusters[p] = newCluster
                changed = True

        if changed: # re-calculate the centroids based on current clustering
            for j in range(k):
                centroids[j] = average_of_points_with_cluster(j)

        # Keep track of best clustering
        if distanceSquaredSum < bestDistanceSquaredSum:
            bestClusters = clusters
            bestDistanceSquaredSum = distanceSquaredSum

```

Voor het kiezen van de random initiële centroids zullen we niet zomaar N-dimensionele punten kiezen. In plaats daarvan nemen we k willekeurige punten uit de input data, waarvoor je de code uit de `Rng` klasse (zie verder) *moet* gebruiken (dan kan iedereen vergelijkbare output bekomen).

## De output

Voor het `mouse_500x2.csv` voorbeeld, met 3 clusters en waarbij er 10 repetitions uitgevoerd werden, ziet de finale output er als volgt uit:

```

# Steps: 11,6,7,6,14,11,8,7,11,7
0,0,1,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ..., 2,1

```

De eerste regel zegt **voor elke repetition hoeveel stappen er uitgevoerd werden** (van random centroids tot finale centroids), in dit geval staan er 10 getallen vermits er 10 repetitions waren. De eerste repetition, waarvoor de evolutie van de centroids in een vorige figuur getoond werd, telde 11 stappen

zoals eerder vermeld. De tweede regel zegt voor elk punt tot welk van de  $k$  clusters het behoort. In dit voorbeeld zijn er 500 punten, en zullen er op deze regel 500 getallen staan. Vermits er  $k=3$  clusters verondersteld werden, is elk van deze getallen een waarde tussen 0 en 2.

Om te helpen tot een correct werkend programma te komen, zijn er ook twee **debug** outputs die aangezet kunnen worden, om de verschillende clusterings tijdens de stappen van één herhaling bij te houden (zoals de plots met 'stap 1' tot 'stap 11'), of om de evolutie van de centroids bij te houden (de plot met '+' en 'o' als begin- en eindpunt). Indien er meerdere herhalingen zijn moeten deze **enkel bij de eerste repetition** bijgehouden worden. Gezien de overhead die dit bovendien met zich meebrengt gebruik je deze debug opties **niet als er tijdsmetingen gebeuren**.

De evolutie van de clusters wordt aangezet met de `--trace` optie, en is enigszins vergelijkbaar met die tweede regel uit de hoofdoutput waar er voor elk punt een getal is die de toegekende cluster aanduidt. Dit keer is het echter niet enkel voor de finale clusters, maar voor alle tussenliggende stappen. In het eerdere `mouse_500x2.csv` voorbeeld waren er 11 stappen, en ziet de bijhorende output er als volgt uit (voor elke stap is er een regel die de clustering beschrijft):

```
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0, ... ,0,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ... ,2,1
0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,1,2, ... ,2,1
```

Met het script `visualize_clusters.py` dat ook verder beschreven wordt, kan je zulke output visualiseren.

Met de `--centroidtrace` optie kan je voor elk van de stappen de centroids bijhouden. Elke centroid is een  $N$ -dimensioneel punt, en er zijn bij elke stap  $k$  centroids. Voor het `mouse_500x2.csv` voorbeeld gaat het om 2D punten, en namen we  $k=3$  centroids. Elke stap zal met deze optie drie lijnen van elk twee coördinaten uitschrijven om zo de bijhorende centroids weer te geven. In dit specifieke voorbeeld waren er 11 stappen en ziet deze debug output er als volgt uit:

```
0.7194189576,0.4364809386
0.7023711449,0.7689249858
0.7116156172,0.817486736
0.5127426065,0.4467446204
0.4820806435,0.7178650368
0.7619422496,0.8096093788
... (nog 3*9 regels, 3 centroids, nog 9 stappen) ...
```

# Startcode

De input en output bestanden zijn in **CSV (comma-separated values)** formaat, en de bestanden `CSVReader.hpp` en `CSVwriter.hpp` bevatten code om zulke bestanden respectievelijk te lezen en te schrijven. Hun gebruik wordt geïllustreerd in het (nog verder in te vullen) hoofdprogramma `main_startcode.cpp`.

Uiteindelijk zijn we geïnteresseerd in de **snelheid** waarmee verschillende implementaties de clusterings kunnen berekenen, en `timer.h` bevat code die hiermee kan helpen. Met de `Timer` klasse zelf kan je de uitvoertijd van het relevante stuk van je code meten. Afhankelijk van de gebruikte methodes kan je in de plaats hiervan ook een andere functie gebruiken, bijvoorbeeld `omp_get_wtime()` voor de OpenMP opdracht, of `MPI_Wtime()` voor de MPI opdracht. De `AutoAverageTimer` klasse kan van pas komen om delen van je code te timen, om te achterhalen waar de **bottlenecks zitten**: voor elke `start()` en `stop()` combinatie wordt de tijd ertussen opgeslagen en uiteindelijk wordt zowel het gemiddelde als de standaardafwijking van al deze intervallen uitgeschreven.

Zoals eerder vermeld is het de bedoeling dat je de `Rng` klasse uit `rng.h` en `rng.cpp` gebruikt om te kiezen welke punten uit de dataset als initiële centroids gebruikt worden. De relevante functie is `pickRandomIndices`. Het **eerste argument** daarvan zegt hoeveel **datapunten** er zijn. Het **tweede argument** is een **vector** die lengte **k (aantal clusters)** moet hebben. De functie zal dan in die vector de (random) indices van punten uit de input data invullen. Wanneer iedereen deze methode gebruikt om de initiële centroids te kiezen, kunnen we zorgen voor echt vergelijkbare resultaten.

De code in `main_startcode.cpp` kan je (maar moet niet) gebruiken om je programma's te maken, en illustreert ook het gebruik van een aantal klassen. De functie `readData` die vermeld wordt illustreert het gebruik van de `CSVReader` klasse om een input CSV bestand in te lezen. Voor het wegschrijven van data wordt steeds de `FileCSVwriter` klasse gebruikt.

De `kmeans` functie bevat de iteratie over het aantal repetitions, maar moet vervolledigd worden om tot een werkend algoritme te komen. Je ziet in deze for-loop ook dat de **debug-files gesloten worden aan het einde van de lus**. Dit is omdat debug output enkel bij de eerste herhaling bijgehouden mag worden.

**Belangrijk** voor alle opdrachten is dat de **timing** van het algoritme steeds **start na het inlezen** van de input data, en **stopt voor het uitschrijven** van de gevonden clusters. De output die naar stdout/stderr geschreven wordt mag je uiteraard aanpassen als een ander formaat of extra output handiger lijkt.

In welke mate je de gegeven code gebruikt bepaal je zelf, maar je programma moet de onderstaande argumenten herkennen, zoals ook in de voorbeeldcode geschetst wordt:

- `--input inputfile.csv`: met deze optie geef je aan welk bestand de inputdata bevat.
- `--output outputfile.csv`: hierin wordt de finale output opgeslagen, zoals eerder beschreven bevat die twee regels.

- `--k aantalclusters` : zo geef je het aantal clusters op waarnaar gezocht moet worden.
- `--repetitions rep` : hiermee geef je het aantal herhalingen op.
- `--seed s` : hiermee geef je een 'seed' op voor de random number generator. Wanneer je dezelfde seed gebruikt, zal je dezelfde resultaten bekomen. **Voor je eigen metingen gebruik je het laagste studentnummer van je groepsleden als seed.**

De opties `--threads T` en `--blocks B` moeten steeds aanvaard worden, maar de interpretatie ervan hangt af van de specifieke opdracht. Ze hebben te maken met de manier waarop er geparalleliseerd wordt. Ze zijn **optioneel aanwezig** op de command line, en hebben **default** waarden van **1**.

Zoals eerder vermeld zijn er twee debug opties, die optioneel zijn op de command line (maar verplicht geïmplementeerd worden):

- `--trace clusters.csv` : schrijf tijdens de eerste herhaling de clusters naar het vermelde CSV bestand, zoals eerder beschreven.
- `--centroidtrace centroids.csv` : schrijf tijdens de eerste herhaling de centroids naar het opgegeven CSV bestand. Ook dit werd eerder toegelicht.

Zet deze **debug opties niet aan wanneer je metingen uitvoert** om na te gaan hoe snel je implementaties zijn.

## Referentieimplementatie

---

Op het VSC vind je een referentieimplementatie, gebaseerd op deze startcode:

```
/data/leuven/303/vsc30380/kmeans_serial_reference
```

Deze is statisch gecompileerd (met de Intel compiler), wat wil zeggen dat je deze executable kan uitvoeren zonder enige modules te moeten laden. Om de resultaten van het `mouse_500x2.csv` voorbeeld te reproduceren, kan je bijvoorbeeld het volgende commando uitvoeren:

```
/data/leuven/303/vsc30380/kmeans_serial_reference --input mouse_500x2.csv \
--output output.csv --repetitions 10 --k 3 --centroidtrace centroidtrace.csv \
--trace clustertrace.csv --seed 1338
```

De code is een rechttoe-rechtaan implementatie van het k-means algoritme; de enige vorm van optimalisatie is het opgeven van `-O3 -DNDEBUG` bij het compileren.

## Hulpscripts

---

Er zijn een aantal scripts die van pas kunnen komen bij het maken van je programma's. De Python 3 scripts gebruiken `pandas` om data in te lezen, `numpy` voor de verwerking en `matplotlib` om resultaten te tonen.

Met `visualize_dataset.py` kan je een opgegeven dataset tonen. De punten uit de dataset hebben vaak meer dan twee coördinaten, vandaar dat je moet opgeven welke coördinaat je moet gebruiken op de x-as en welke op de y-as (beide zijn indices met betrekking tot de kolom die gebruikt moet worden, en beginnen dus bij 0). Het gebruik ervan is als volgt:

```
./visualize_dataset.py input.csv xcol ycol
```

Het `visualize_clusters.py` script kan gebruikt worden zowel om de finale clusters te tonen, als om eventueel tussentijdse clusters te tonen uit een bestand verkregen met de `--trace` optie. In het eerste geval wordt er één enkele plot getoond, in het tweede geval wordt de volgende clustering geplott wanneer je de huidige plot sluit. Ook hier moet je opgeven welke kolom als x-coördinaat dient en welke als y-coördinaat. Het gebruik is als volgt:

```
./visualize_clusters.py input.csv clustertrace.csv xcol ycol
```

De output van `--centroidtrace` kan je visualiseren met het `./visualize_centroids.py` script. Bij dit script moet je ook opgeven hoeveel centroids er zijn, vermits dat niet automatisch afgeleid kan worden uit het bestand zelf. Je kan dit script als volgt gebruiken:

```
./visualize_centroids.py input.csv centroidtrace.csv k xcol ycol
```

Het `compare.py` script dient om de outputs van twee versies van je kmeans programma met elkaar te vergelijken. Wanneer voor een bepaalde input file het aantal clusters, het aantal repetitions en de seed hetzelfde zijn, dan moet de output steeds hetzelfde zijn. Met dit script worden automatisch twee executables uitgevoerd met dezelfde settings, en de verkregen output wordt met elkaar vergeleken. Het gebruik ziet er als volgt uit:

```
./compare.py ./exe1 ./exe2 ...
```

Na het opgeven van de twee executables volgen dezelfde command line arguments die eerder vermeld werden. Bij de output files wordt er een '.1' of '.2' achter de opgegeven filename gezet, afhankelijk van de executable die uitgevoerd werd. Zowel de hoofdoutput als de debug outputs kunnen zo vergeleken worden.

**Om bijvoorbeeld jouw seriële implementatie te vergelijken met de referentieimplementatie, kan je het volgende commando uitvoeren:**

```
./compare.py ./my_serial_kmeans /data/leuven/303/vsc30380/kmeans_serial_reference \  
--input mouse_500x2.csv --output out.csv --k 3 --repetitions 10 --seed 12345 \  
--trace clusters.csv --centroidtrace centroids.csv
```



Je zal dan zien dat de verschillende output files met elkaar vergeleken worden, en er een foutmelding uitgeschreven wordt indien ze niet overeenstemmen.

Als je steeds met dezelfde output wil vergelijken heeft het weinig zin om bijvoorbeeld de referentieimplementatie steeds opnieuw uit te voeren. Als je de output van een van de executables al hebt en je wil de executable niet opnieuw uitvoeren, kan je in plaats van de naam van de executable ook 'SKIP' vermelden.

## Opdracht 1: seriële implementatie

---

Als eerste opdracht maak je een volledig seriële implementatie van het k-means algoritme. Voor dezelfde settings moet de output hetzelfde zijn als die van de referentie-implementatie. Indien je 'Eigen' wenst te gebruiken voor deze of volgende implementaties, dan mag dit (maar moet zeker niet), maar let er dan op dat je bij de compilatie van je programma `-DEIGEN_DONT_PARALLELIZE` opgeeft zodat Eigen zelf geen parallelisatie probeert toe te voegen, en we daar zelf volledig controle over hebben. Voor deze versie worden de opgegeven waarden bij de `--threads` en `--blocks` argumenten gewoon genegeerd.

Wanneer je het k-means algoritme onder de loep neemt, zie je dat er op voorhand al geweten is hoeveel geheugen er nodig is om het algoritme tot een goed einde te brengen. Strikt genomen zijn er dan ook geen dynamische geheugenallocaties nodig van zodra het algoritme bezig is. Hou dit in gedachten wanneer je je algoritme schrijft: dynamische allocaties (die bijvoorbeeld ook intern in een 'vector' gebruikt worden) kosten tijd, en afhankelijk van waar ze precies in je algoritme voorkomen kan dit een grote impact hebben. Het is niet noodzakelijk dat je zulke dynamische allocaties helemaal mijdt, maar wel deze die een merkbaar effect hebben op de uitvoertijd.

Wat je bij deze opdracht moet indienen, is een zip file met daarin:

- de broncode van je correct werkende implementatie (deze moet dus dezelfde output produceren als de referentieimplementatie)
- een eenvoudige Makefile (cfr Makefile bij de startcode) om deze te bouwen
- een tekstfile waarin je testsettings vermeldt (input file, seed, etc), de tijd die jouw implementatie erover deed, en de tijd die de referentieimplementatie nodig had. Een enkele meting volstaat hier, maar je mag natuurlijk ook je metingen herhalen en gemiddelde en standaardafwijking vermelden. Doe de meting op een VSC rekennode, en **niet op de login node**. **Compileer je code zeker met optimalisaties** (gebruik minstens `-DNDEBUG -O3`), en gebruik de debug flags niet wanneer je je metingen uitvoert.

De naam van deze zip file is `serial_xxxxxxx.zip`, waarbij je de `xxxxxxx` vervangt door het laagste studentnummer van je groepsleden.

---

## Algemene opmerkingen bij parallelle implementaties

---

We zullen achtereenvolgens OpenMP, CUDA en MPI gebruiken om parallelisatie van het k-means algoritme te bestuderen. Het is echter niet de bedoeling om deze technologieën door elkaar te gebruiken (bvb geen OpenMP en MPI combineren).

Afhankelijk van de manier waarop je zaken paralleliseert, kan het zijn dat floating point operaties in een andere volgorde worden uitgevoerd. Vermits floating point operaties niet associatief zijn, kan dit voor lichtjes andere centroids zorgen, wat op zijn beurt tot andere clusterings kan leiden. Dit zou dan weer tot meer of minder stappen kunnen leiden binnen een bepaalde repetitie.

Vermits we willen weten hoeveel sneller een algoritme wordt met bijvoorbeeld verschillend aantal threads, is het wel belangrijk dat dezelfde berekeningen gebeuren, anders heeft de vergelijking weinig zin. Je gebruikt voor je metingen uiteraard dezelfde settings (input data, aantal clusters, aantal herhalingen en seed), maar **gebruik ook enkel je gemeten tijd wanneer dezelfde output bekomen wordt als in je seriële implementatie**. Dit is overigens ook de reden om die eerste regel met aantal stappen per herhaling in de output op te nemen: wanneer zowel dit als de gevonden clustering hetzelfde is, kunnen we er erg zeker van zijn dat dezelfde berekeningen werden uitgevoerd (of op zijn minst een equivalente hoeveelheid).

Voor de volgende opdrachten maak je gebruik van een Google Colab om je antwoorden op de opgegeven vragen neer te schrijven. Hiermee maak je duidelijk in welke mate je inzicht hebt in de gebruikte methodes, en het is dan ook van groot belang. Zoals in de theorie wordt aangehaald, maak je op je figuren ook steeds duidelijk wat er geplot wordt (assen labelen, eenheden vermelden). Je bevindingen baseer je ook nooit op één enkele meting, maar je zorgt voor herhalingen zodat je een gemiddelde en standaardafwijking kan bepalen (of bvb mediaan en kwartielen).

Alle **metingen** voor deze Colabs moeten gebeuren op **rekennodes van het VSC**, dus niet op een login node, en niet op een van de debug nodes.

## Opdracht 2: parallelisme met OpenMP

---

Voor deze opdracht gebruik je OpenMP om meerdere cores van een node te gebruiken om versnelling van het k-means algoritme te verkrijgen. Het aantal zulke threads dat gebruikt moet worden, specificeer je met de `--threads` optie; de waarde bij `--blocks` wordt genegeerd.

Je kiest zelf een dataset (mag ook een andere zijn dan de opgegeven datasets), en waarden voor  $k$  en voor het aantal repetities (zorg wel dat  $k < 100$  en **repetitions**  $< 200$ ), die steek houden met de manier waarop je je parallelisatie inbouwt. We **houden deze waarden verder vast**, we zullen bvb niet kijken naar uitvoertijd voor verschillende waarden van  $k$ .

Je hoeft in je Colab niet meer uit te leggen wat k-means is, maar beantwoord de volgende vragen:

1. Schets je werkwijze, maw leg uit wat je geparallelliseerd hebt en waarom. Zijn er delen die je bewust niet geparallelliseerd hebt (en waarom niet)? Hoe past dit bij de keuze van je dataset en parameters ( $k$ , repetitions)?

2. Maak een vergelijking van je seriële uitvoertijd met die van je OpenMP implementatie voor 1 thread. Baseer dit op een aantal herhaalde metingen.
3. Toon (figuren!) hoe de efficiëntie *en* speedup veranderen met het gebruikte aantal threads/cores. Denk er weer aan dat je je metingen herhaalt zodat je bvb gemiddelde en standaardafwijking kan tonen, en vermeld dit aantal. Gebruik de gemiddelde uitvoertijd van de 1-thread OpenMP versie als referentietijd. Bespreek de schaalbaarheid van je implementatie.
4. Deze metingen kan je gebruiken (zoals in de theorie) om de seriële fractie te schatten voor een bepaald aantal threads. Gebruik de metingen voor 8 threads, en schat zo de seriële fractie (de gemiddelde tijden gebruiken volstaat, bvb standaardafwijking hoeft niet).
5. Op basis van deze seriële fractie kan je dan weer de uitvoertijd schatten voor 16 threads. Doe dit, vergelijk met je gemeten uitvoertijd voor 16 threads, en bespreek het verschil.
6. Gebruik je metingen/berekeningen om je verwachtingen te beschrijven als je meer en meer cores ter beschikking krijgt. Wat als dit oneindig veel zou zijn? Wat wordt de uitvoertijd in dat geval?
7. Welke manieren kan je bedenken om de schaalbaarheid nog te verbeteren (nog steeds met enkel OpenMP)? Hoeveel werk zouden ze vergen?

Wat je bij deze opdracht moet indienen, is een zip file met daarin:

- de broncode van je correct werkende implementatie
- een eenvoudige Makefile (cfr Makefile bij de startcode) om deze te bouwen
- de gedownloade Colab (.ipynb file) met naam `openmp_XXXXXXX.ipynb` (de X'en zijn weer je studentnummer)
- als je een gekende dataset gebruikte dien je die niet terug in, als je een andere dataset gebruikte wel
- de ruwe metingen (bvb CSV, of Excel) die je gebruikt hebt, eventuele scripts die je gebruikte om ze te verwerken
- je pbs bestanden

De naam van deze zip file is `openmp_XXXXXXX.zip`.

## Opdracht 3: parallelisme met CUDA

---

Ook voor de CUDA versie willen we bestuderen hoe een oplossing schaalbaar is wanneer we meer rekeneenheden gebruiken. De onafhankelijke rekeneenheden op NVIDIA GPUs zijn de Streaming Multiprocessors (SMs), en de enige manier om controle uit te oefenen over hoe die ingezet worden is door de keuze van het aantal blocks. In deze opdracht zal je zelf kunnen kiezen hoeveel threads per block er zijn, en zullen we de uitvoertijd meten voor een verschillend aantal blocks.

Dit wil zeggen dat je implementatie **niet gewoon één CUDA thread per datapunt** kan uitvoeren, en het benodigd aantal blocks daarvoor berekent op basis van het aantal datapunten. Vermits je programma moet werken met 1, 2, ... blocks, zal een CUDA thread typisch meer dan één datapunt behandelen.

Concreet maak je een implementatie waarbij de `--blocks` command line parameter aangeeft hoeveel blocks er gebruikt mogen worden, en `--threads` hoeveel threads per block. Bij je tests bepaal je die laatste zelf: gebruik het aantal threads per block (veelvoud van 32) wat betere resultaten geeft. Dit aantal threads per block hou je verder vast (uiteraard vermeld je in je Colab waarvoor je koos), je laat voor je metingen het aantal blocks variëren.

Afhankelijk van hoeveel werk je op de GPU doet, en wat de CPU nog zelf moet doen, kan het zijn dat het bijhouden van debug logs voor clusters en centroids erg onpraktisch wordt. Wanneer dit zo is, hoeft je deze debug opties niet te voorzien in je code. Vergewis wel jezelf ervan tijdens het ontwikkelen dat je code doet wat het moet doen, dat de berekende clusters en centroids correct zijn, bvb door het gebruik van enkele `printf` statements (die je achteraf uiteraard niet meer gebruikt) in je kernels.

Ook afhankelijk van je aanpak kan het zijn dat je een typische operatie zoals een reductie nodig hebt. Hiervoor is het toegestaan om de `thrust` library te gebruiken. Merk op dat je bij zulke calls niet kunt aangeven hoeveel blocks of threads per block er gebruikt moeten worden, hiermee moet je dus voor `thrust` calls geen rekening houden.

Je aanpak hoeft overigens niet hetzelfde te zijn als bij OpenMP. Kies ook hier weer een dataset en parameters die passen bij je aanpak.

De vragen die je moet beantwoorden zijn gelijkaardig aan die van OpenMP, alleen variëren we nu het aantal blocks:

1. Schets je werkwijze, maw leg uit wat je geparallelliseerd hebt en waarom. Zijn er delen die je bewust niet geparallelliseerd hebt (en waarom niet)? Hoe past dit bij de keuze van je dataset en parameters (k, repetitions)?
2. Maak een vergelijking van je seriële uitvoertijd met die van je CUDA implementatie voor 1 block. Baseer dit op een aantal herhaalde metingen.
3. Toon (figuren!) hoe de efficiëntie *en* speedup veranderen met het gebruikte aantal blocks. Denk er weer aan dat je je metingen herhaalt zodat je bvb gemiddelde en standaardafwijking kan tonen, en vermeld dit aantal. Gebruik de gemiddelde uitvoertijd van de 1-block CUDA versie als referentietijd (dus niet de seriële versie in dit geval). Bespreek de schaalbaarheid van je implementatie.
4. Deze metingen kan je gebruiken (zoals in de theorie) om de seriële fractie te schatten voor een bepaald aantal threads. Gebruik de metingen voor 8 blocks, en schat zo de seriële fractie (de gemiddelde tijden gebruiken volstaat, bvb standaardafwijking hoeft niet).
5. Op basis van deze seriële fractie kan je dan weer de uitvoertijd schatten voor 16 blocks. Doe dit, vergelijk met je gemeten uitvoertijd voor 16 blocks, en bespreek het verschil.
6. Gebruik je metingen/berekeningen om je verwachtingen te beschrijven als je meer en meer SMS ter beschikking krijgt. Wat als dit oneindig veel zou zijn? Wat wordt de uitvoertijd in dat geval?
7. Welke manieren kan je bedenken om de schaalbaarheid nog te verbeteren (nog steeds met enkel CUDA)? Hoeveel werk zouden ze vergen?

Wat je bij deze opdracht moet indienen, is een analoge zip file als bij de OpenMP opdracht, met naam `cuda_XXXXXX.zip`.

## Opdracht 4: parallellisme met MPI

Met MPI kunnen we meerdere nodes gebruiken om berekeningen te paralleliseren. Ook binnen één node kan dit gebruikt worden, maar we willen ons niet tot één enkele node beperken. Voor deze opdracht gebruik je maximaal 4 nodes tegelijk op het VSC. Geef duidelijk aan bij je tests hoeveel nodes je gebruikt en hoeveel cores per node, vermits er meerdere mogelijkheden zijn.

Bij deze MPI versie mag enkel het proces met rang 0 bestanden lezen en schrijven.

Voor deze opdracht wordt het `--blocks` argument helemaal genegeerd. Omdat je je programma met `mpirun` moet starten, kan je in je programma zelf eigenlijk geen gebruik meer maken van `--threads` om een aantal processen op te geven, je kan enkel nog nagaan hoeveel er gestart werden met `MPI_Comm_size`. Eventueel kan het `mpiwrapper.sh` script van pas komen: dit script vertaalt een `--threads` argument in een aantal processen en start een `mpirun` commando. De executable die uitgevoerd moet worden geef je dan aan met de `EXECUTABLE` environment variable, bvb

```
EXECUTABLE=./kmeans_mpi ./mpiwrapper.sh --threads 8 --input mouse_500x2.csv ...
```

wordt vertaald in

```
mpirun -n 8 ./kmeans_mpi --threads 8 --input mouse_500x2.csv ...
```

Je kan dit script op deze manier ook samen gebruiken met het `compare.py` script (voer `mpiwrapper.sh` uit zonder argumenten om een voorbeeldje te zien).

Je aanpak hoeft overigens niet hetzelfde te zijn als bij OpenMP of CUDA. Kies ook hier weer een dataset en parameters die passen bij je aanpak.

De vragen die je moet beantwoorden zijn gelijkaardig aan die van OpenMP en CUDA, maar we variëren nu het aantal processen:

1. Schets je werkwijze, maw leg uit wat je geparallelliseerd hebt en waarom. Zijn er delen die je bewust niet geparallelliseerd hebt (en waarom niet)? Hoe past dit bij de keuze van je dataset en parameters (k, repetitions)?
2. Maak een vergelijking van je seriële uitvoertijd met die van je MPI implementatie voor 1 proces, en met je OpenMP versie voor 1 thread. Baseer dit op een aantal herhaalde metingen.
3. Toon (figuren!) hoe de efficiëntie *en* speedup veranderen met het gebruikte aantal processen. Doe dit **voor 2 scenarios**: één waarbij alle processen op dezelfde node runnen, en één waarbij eenzelfde aantal processen over 4 nodes verdeeld worden (vermeld zeker hoe de verdeling gebeurt). Is er een verschil (en verklaar waarom wel of niet)? Denk er weer aan dat je je metingen herhaalt. Gebruik de gemiddelde uitvoertijd van een 1-proces MPI versie als referentietijd, en bespreek de schaalbaarheid van je implementatie.
4. Deze metingen kan je gebruiken (zoals in de theorie) om de seriële fractie te schatten voor een bepaald aantal processen. Gebruik je metingen voor 4 nodes, en voor 8 processen, en schat zo

de seriële fractie (de gemiddelde tijden gebruiken volstaat, bvb standaardafwijking hoeft niet).

5. Op basis van deze seriële fractie kan je dan weer de uitvoertijd schatten voor 16 processen. Doe dit, vergelijk met je gemeten uitvoertijd voor 16 processen, en bespreek het verschil. (Ook hier gebruik je je metingen voor 4 nodes).
6. Gebruik je metingen/berekeningen om je verwachtingen te beschrijven als je meer en meer nodes en cores ter beschikking krijgt. Wat als dit oneindig veel zou zijn? Wat wordt de uitvoertijd in dat geval?
7. Welke manieren kan je bedenken om de schaalbaarheid nog te verbeteren (nog steeds met enkel MPI)? Hoeveel werk zouden ze vergen?

Wat je bij deze opdracht moet indienen, is gelijkaardig als bij de vorige opdrachten, nu met naam `mpi_XXXXXXX.zip`.

## Opdracht 5: optionele terugblik

---

Aan je laatste Colab mag je (moet niet, is optioneel) een sectie toevoegen waarin je terugblijkt op deze practica. Hoeveel tijd heeft het programmeren in beslag genomen? Hoeveel tijd de metingen/besprekingen? Lag de moeilijkheidsgraad te hoog of te laag? Zijn er zaken die je zou aanpassen? Je kan andere opmerkingen over het project ook hier vermelden.