Time Series Similarity Search

Practice lab — Data Acquisition, Extraction and Storage
Master 2 (PSL)

ATTENTION!

Every computation depends on your **personal seed** u_0.
Write it in the variable `SEED` of your notebook.
But **first** run everything with the default seed $\widetilde{u_0} = 42$
and compare with the reference sheet (Appendix A).

Organisation. Sections 1 to 3 are done **in class**; Sections 4 and 5 are done **at home**. You hand in a single Jupyter notebook (the template `lab_student.ipynb` is provided). The lab contains two kinds of questions:

- **Questions Q** are *numerical*: they have four items a)–d) whose values depend on your seed. Fill the dictionary `ANSWERS` at the end of the notebook. They are graded automatically.
- **Questions T** are *theoretical or experimental*: answer them in the Markdown cells of the notebook. Be short and precise: a proof, an algorithm described in a few sentences or lines of pseudo-code (do not paste your code), a complexity, a plot with a comment. When a complexity is asked, give the order of magnitude as a function of the parameters (e.g. $O(nl)$, $O(l \log l)$).

Allowed libraries. The Python standard library (`math`, `heapq`, `bisect`, `collections`, `itertools`, `fractions`, `time`, ...), `numpy` and `matplotlib`. Vectorising with `numpy` is encouraged. You must **not** use any library that directly answers a question: no `scipy.spatial`, `sklearn.neighbors`, `tslearn`, `sktime`, `aeon`, `pyts`, `dtaidistance`, `fastdtw`, `stumpy`, `rtree`, `faiss`, etc. Every distance, representation and index is implemented by you.

Efficiency. Most numerical questions are designed so that a naive algorithm is too slow on items c)–d). With a good algorithm, every item runs in less than a minute in plain Python/numpy. Start with small parameters, and check your programs on small examples solved by hand or with the reference sheet.

Floating-point numbers. All the series of this lab take **integer values** (on purpose): squared Euclidean distances, squared DTW distances, etc. are integers, and you may exploit it to compute and compare quantities exactly. Unless stated otherwise:

- real-valued answers are given **rounded to 2 decimals** (they are accepted up to ± 0.01, or a relative error of 10^{-5} for large values);
- when a question asks to *count* elements satisfying an inequality, the inequality is **strict**. Borderline cases that depend on rounding errors are detected and handled by the grader.

1 Preliminaries

We study *similarity search* in time-series databases: given a database D of n series and a query series Q, find the series of D that is most similar to Q. This requires (i) a similarity measure that is meaningful and (ii) a way to avoid comparing Q to every series of D with an expensive measure. Both aspects are covered in this lab.

1.1 Notations

A time series of length l is a vector $T = [t_0, \dots, t_{l-1}] \in \mathbb{R}^l$; we write $|T| = l$. A database of n series of length l is $D_{n,l} = [T_0, \dots, T_{n-1}]$. Given a distance d, the *nearest neighbour* of Q in D is

$$NN_d(Q) = \arg\min_{T \in D} d(Q, T).$$

In this lab the nearest neighbour is always unique for the default seed (and almost surely for yours).

1.2 The dataset

The notebook provides the (short) generator code; do **not** modify it. It relies on the linear congruential generator

$$u_{k+1} = (1\,103\,515\,245 \times u_k + 12\,345) \bmod 2^{31}$$

seeded from your u_0. Each series is a slow integer-valued random walk to which one to three rectangular "bumps" are added. We use the following notations:

- `series(SEED, k, l)` returns the series $T_k^{(l)}$ of length l (integer values). **Warning:** for each length l the series is generated independently: $T_k^{(50)}$ is *not* the prefix of length 50 of $T_k^{(256)}$;
- `database(SEED)` returns the database $D = D_{1000,256}$ as a 1000×256 array; its row k is $T_k = T_k^{(256)}$. *When no length is specified, T_k denotes the row k of D;*
- `query(SEED, j, D)` returns the query Q_j ($j \in \{0, 1, 2, 3\}$), a series of length 256. Have a look at how the queries are built: it will help you interpret your results.

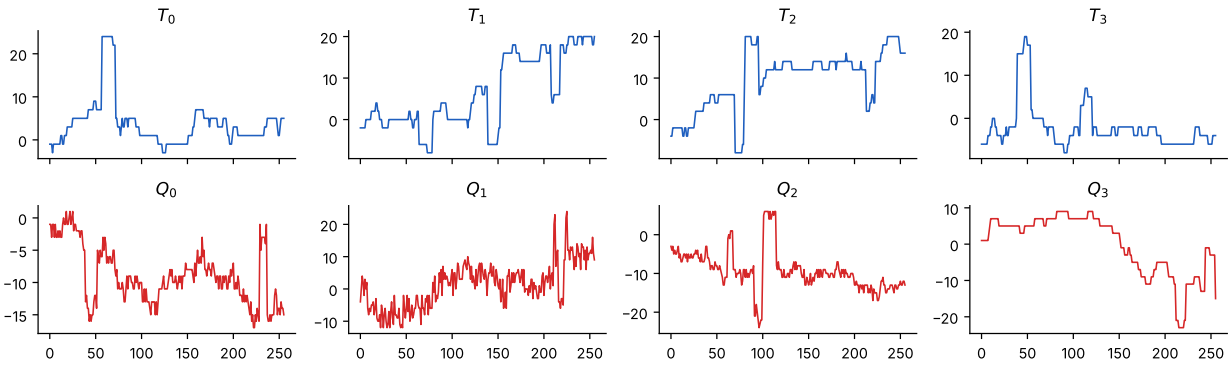


Figure 1: The series $T_0, \dots, T_3$ of the database (top) and the four queries (bottom), for $\widetilde{u_0} = 42$.

Question Q1 (in class) *Generate the database D and the queries. For the following series T_k of D, give the amplitude $\max_i t_i - \min_i t_i$:*

a) $k = 0$ **b)** $k = 1$

c) $k = 500$ **d)** $k = 999$

2 Euclidean distance, PAA and lower-bounding search (in class)

2.1 Euclidean distance

The Euclidean distance between two series of the same length l is

$$\mathsf{ED}(T_a, T_b) = \sqrt{\sum_{k=0}^{l-1} (t_{a,k} - t_{b,k})^2}.$$

Question Q2 *Give* $\mathsf{ED}(T_a, T_b)$ *for the following pairs:*

a) T_0, T_1 **b)** T_{10}, T_{20}

c) T_{100}, T_{200} **d)** T_{998}, T_{999}

2.2 Piecewise Aggregate Approximation

For long series, computing distances is expensive. A classical idea is to compute a compact representation of every series and to use it to discard candidates cheaply. For N dividing $|T| = l$, the PAA of order N of T is $\widetilde{T} = [x_0, \ldots, x_{N-1}]$ where x_i is the mean of T over its i-th segment of l/N points:

$$x_i = \frac{N}{l} \sum_{j=\frac{l}{N}i}^{\frac{l}{N}(i+1)-1} t_j.$$

Between two PAA representations of the same order, we define

$$\mathsf{DR}(\widetilde{T}_a, \widetilde{T}_b) = \sqrt{\frac{l}{N} \sum_{i=0}^{N-1} (x_{a,i} - x_{b,i})^2}.$$

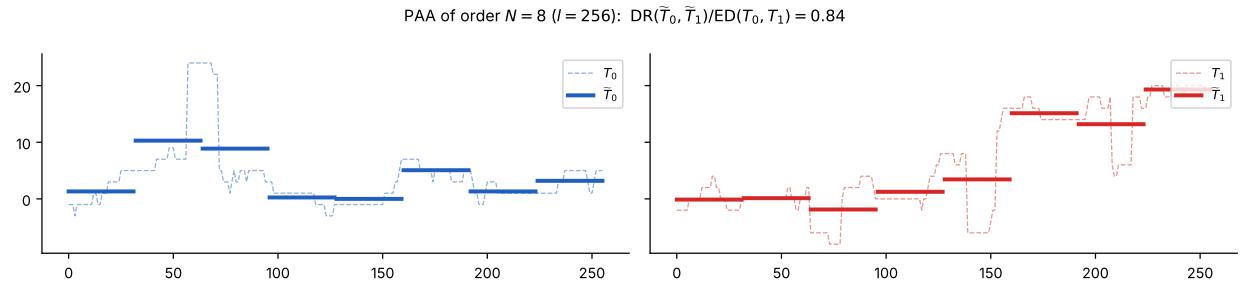


Figure 2: PAA representations of T_0 and T_1 ($\widetilde{u_0} = 42$).

Question T1 *Prove that* DR *is a lower bound of* ED*: for all* $T_a, T_b \in \mathbb{R}^l$, $\mathsf{DR}(\widetilde{T}_a, \widetilde{T}_b) \leq \mathsf{ED}(T_a, T_b)$.

Question T2 *Let* $N \mid N'$ *and* $N' \mid l$*. Compare* DR *computed with PAA of order* N *and of order* N'*, and prove your claim. What is* DR *when* $N = l$*?*

Question Q3 *Give* $\mathsf{DR}(\widetilde{T}_0, \widetilde{T}_1)$ *for the following PAA orders:*

a) $N = 4$ **b)** $N = 16$

c) $N = 64$ **d)** $N = 256$

2.3 Nearest neighbour search

Question Q4 *By brute force, give the nearest neighbour* $NN_{\mathsf{ED}}(Q)$ *(its index* k*) and the distance* $\mathsf{ED}(Q, T_k)$*, for:*

a) $Q = Q_0$ **b)** $Q = Q_1$
c) $Q = Q_2$ **d)** $Q = Q_3$

Answer format: `(k, distance)`.

We now pre-compute the PAA (order N) of all the series of D, and want to use it to avoid as many exact ED computations as possible, while still returning the *exact* nearest neighbour.

Question T3 *Propose an algorithm* findNN *that returns the exact* $NN_{\mathsf{ED}}(Q)$ *using the PAA representations, and that has a time complexity of* $O(nN + l)$ *in the best case. Explain why it never misses the nearest neighbour. What is its worst-case complexity as a function of* n*,* N *and* l*, and when does the best case happen?*

Question Q5 *Give the number of series* $T \neq NN_{\mathsf{ED}}(Q)$ *of* D *such that*

$$\mathsf{DR}(\widetilde{Q}, \widetilde{T}) < \mathsf{ED}(Q, NN_{\mathsf{ED}}(Q)),$$

where the PAA is of order N*, for:*

a) $Q = Q_2$, $N = 4$ **b)** $Q = Q_3$, $N = 2$
c) $Q = Q_3$, $N = 4$ **d)** $Q = Q_3$, $N = 8$

Question T4 (experiment) *What is the link between the number computed in the previous question and the number of exact distance computations performed by your* findNN*? Plot, for each query, the number of exact* ED *computations and the running time of* findNN *as functions of* $N \in \{2, 4, \ldots, 128\}$*, and compare with the brute force. Comment on the trade-off.*

3 An elastic distance: Dynamic Time Warping (in class)

The Euclidean distance compares the series point by point: two series that are similar up to small temporal shifts can be far apart (Figure 3(a)). Dynamic Time Warping (DTW) instead looks for the best *alignment* between the points of the two series (Figure 3(b)).

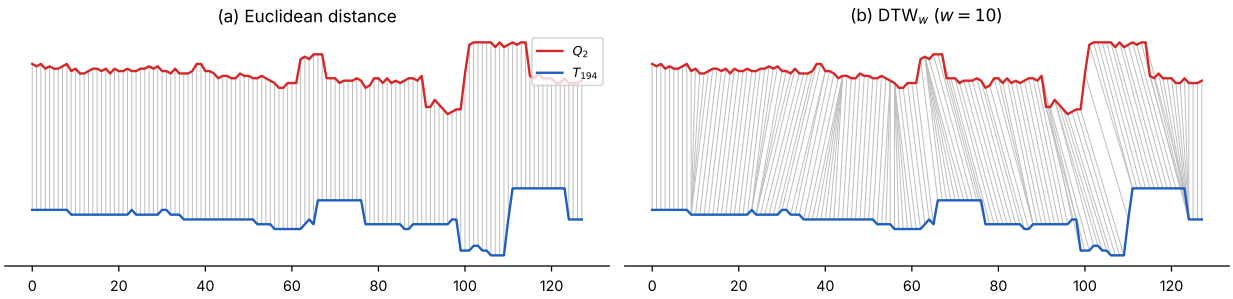


Figure 3: Point-to-point matching of ED (a) and alignment of DTW (b) between the first 128 points of Q_2 and of a database series ($\widetilde{u_0} = 42$).

For two series T_a and T_b of lengths l_a and l_b (possibly different),

$$\mathsf{DTW}(T_a, T_b) = \min_{\pi \in \mathcal{A}(T_a, T_b)} \sqrt{\sum_{(i,j) \in \pi} (t_{a,i} - t_{b,j})^2},$$

where a *warping path* $\pi = ((i_0, j_0), \ldots, (i_{K-1}, j_{K-1}))$ is a sequence of K pairs of indices, and $\mathcal{A}(T_a, T_b)$ is the set of admissible paths, i.e. satisfying:

(P1) the beginnings and the ends of the series are matched: $\pi_0 = (0, 0)$ and $\pi_{K-1} = (l_a - 1, l_b - 1)$;

(P2) the path is monotone and continuous: $i_{k-1} \leq i_k \leq i_{k-1} + 1$ and $j_{k-1} \leq j_k \leq j_{k-1} + 1$ for all k (in particular every index of T_a and of T_b appears in π);

(P3) every pair (i, j) appears at most once in π.

Question T5 *What is the order of magnitude of the worst-case time complexity of the trivial algorithm that evaluates every admissible path of $\mathcal{A}(T_a, T_b)$? (A lower bound on $|\mathcal{A}(T_a, T_b)|$ is enough to conclude.)*

Question T6 *Design a polynomial algorithm computing $\mathsf{DTW}(T_a, T_b)$. Justify its correctness and give its time and memory complexities in the worst case as functions of l_a and l_b. Can the memory be reduced if only the distance (and not the path) is needed?*

Question Q6 *Implement your algorithm and give $\mathsf{DTW}(T_a, T_b)$ for:*

a) $T_1^{(50)}, T_2^{(60)}$ **b)** $T_{10}^{(300)}, T_{20}^{(200)}$

c) $T_{30}^{(1000)}, T_{40}^{(1500)}$ **d)** $T_{50}^{(3000)}, T_{60}^{(3000)}$

Question T7 *Is DTW a metric? Justify (if not, give the smallest counter-examples you can). Why does it matter for similarity search?*

Question T8 *Propose a lower bound of $\mathsf{DTW}(T_a, T_b)$, valid for series of any lengths, that can be computed in $O(1)$ time (possibly after an $O(l)$ pre-processing of each series, done once when it is inserted in the database). Prove that it is a lower bound.*

3.1 A constrained elasticity

The optimal path can correspond to extreme temporal distortions, which are often meaningless. A common constraint is the *Sakoe–Chiba band* (Figure 4). **For series of the same length** l, the DTW_w distance (band of width w) is defined as DTW, but only the admissible paths such that $|i - j| \leq w$ for all $(i, j) \in \pi$ are considered.

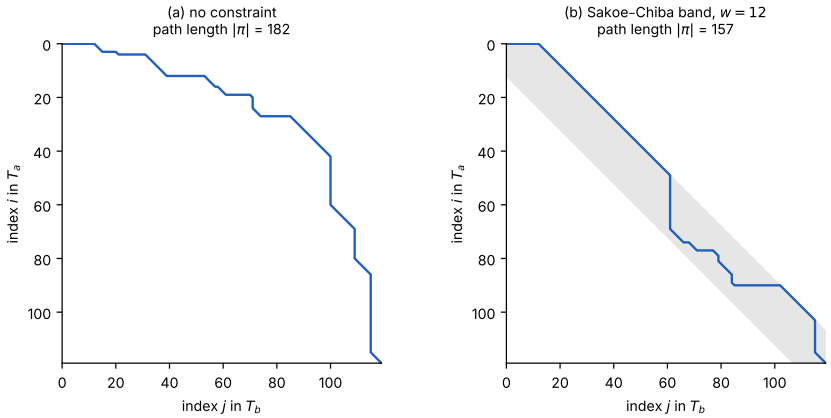


Figure 4: Optimal warping paths without constraint (a) and within a Sakoe–Chiba band (b).

Question T9 *What are DTW_0 and DTW_{l-1}? Show that $w \mapsto \mathsf{DTW}_w(T_a, T_b)$ is non-increasing and deduce an ordering between ED, DTW_w and DTW. Give the time and memory complexities of your implementation of DTW_w as functions of l and w.*

Question Q7 *Give $\mathsf{DTW}_w(T_a, T_b)$ for:*

a) T_0, T_1 and $w = 0$ **b)** T_0, T_1 and $w = 10$

c) T_0, T_1 and $w = 255$ **d)** $T_5^{(10^5)}, T_6^{(10^5)}$ and $w = 20$

Question Q8 *Give the nearest neighbour* $NN_{\mathsf{DTW}_w}(Q)$ *(index* k*) and* $\mathsf{DTW}_w(Q, T_k)$ *for:*

a) $Q = Q_1$, $w = 10$ **b)** $Q = Q_2$, $w = 10$

c) $Q = Q_3$, $w = 10$ **d)** $Q = Q_2$, $w = 5$

Answer format: `(k, distance)`.

Question T10 (experiment) *With* $w = 10$*, compare the nearest neighbours of the four queries under* ED *(Question Q4) and under* DTW_w*. For each query, compare the ratio between the distances of the second and of the first nearest neighbour, under* ED *and under* DTW_w*. What changes for* Q_2 *when* $w = 5$*? Interpret the results in the light of the way the queries are generated.*

4 Adaptive Piecewise Constant Approximation (at home)

The PAA uses segments of equal length, which may hide important features (e.g. short bumps). The APCA (*Adaptive Piecewise Constant Approximation*) uses N segments of *variable* lengths, chosen to minimise the approximation error. Formally,

$$\mathsf{APCA}(T) = \{(x_i, [b_i, e_i])\}_{i=0}^{N-1}, \qquad b_0 = 0, \quad e_{N-1} = l - 1, \quad e_i = b_{i+1} - 1, \quad x_i = \frac{1}{e_i - b_i + 1} \sum_{j=b_i}^{e_i} t_j.$$

For a segment $s = (x_s, [b_s, e_s])$ of length $l_s = e_s - b_s + 1$, its approximation error is

$$\mathsf{Err}(s, T) = \sum_{j=b_s}^{e_s} (t_j - x_s)^2,$$

and the error of an APCA is the sum of the errors of its segments.

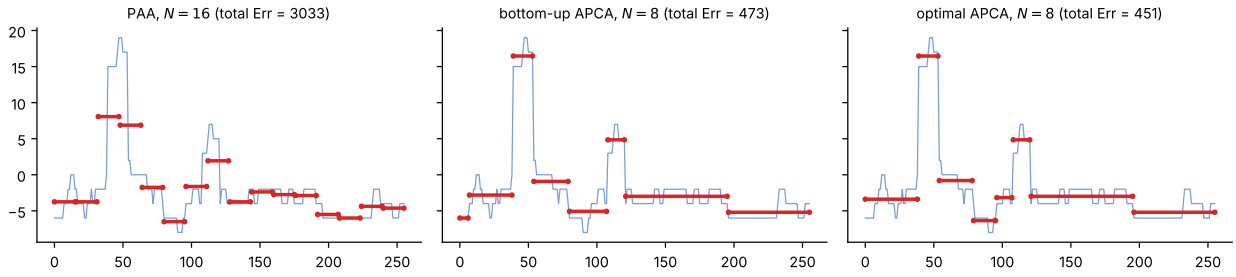


Figure 5: PAA vs. APCA with 16 and 8 segments (16 numbers for both) of T_3 ($\widetilde{u_0} = 42$).

4.1 Computing errors efficiently

Question T11 *Show how to compute* $\mathsf{Err}(s, T)$ *in* $O(1)$ *for any segment* s*, after an* $O(l)$ *preprocessing of* T*. Which numerical problems can arise if this is done naively with floating-point numbers on long series, and how can you avoid them here?*

Question Q9 *Implement* $\mathsf{computeMinimalCost}(T, l_s)$ *returning the start* $b_{\min}$ *of a segment of length* l_s *of minimal error, and this minimal error. Give* $(b_{\min}, \mathsf{Err})$ *for (if several segments are optimal, any of them is accepted):*

a) $T_0^{(10^3)}$, $l_s = 50$ **b)** $T_5^{(10^4)}$, $l_s = 500$

c) $T_{10}^{(10^5)}$, $l_s = 5000$ **d)** $T_{20}^{(10^6)}$, $l_s = 10^4$

4.2 A greedy approach: bottom-up APCA

The *bottom-up* algorithm ($\mathsf{BuAPCA}(T, N)$) starts from the finest segmentation (l segments of length 1), and repeatedly merges the pair of *consecutive* segments whose merged segment has the smallest error Err, until N segments remain. For the results to be reproducible:

- the cost of merging two consecutive segments is the error Err of the merged segment;
- if several pairs have the same minimal cost, the **leftmost** pair (the one whose first segment starts at the smallest index) is merged;
- costs must be compared **exactly**. Hint: $l_s \cdot \mathsf{Err}(s, T)$ is an integer.

Question Q10 *Implement BuAPCA. Give the length of the longest segment and the total error of the APCA for:*

a) $T_0^{(1000)}$, $N = 10$ **b)** $T_5^{(10^4)}$, $N = 20$

c) $T_{10}^{(2\cdot 10^4)}$, $N = 50$ **d)** $T_{20}^{(10^5)}$, $N = 100$

Answer format: `(longest, total error)`.

Question T12 *What is the worst-case time complexity of a straightforward implementation of BuAPCA? Describe a strategy (data structures) achieving $O(l \log l)$.*

4.3 An optimal solution

An APCA with N segments is *optimal* if its total error is minimal among all segmentations of T into N segments.

Question T13 *Design an algorithm computing an optimal APCA. Justify it and give its time and memory complexities as functions of l and N. Compare with the trivial algorithm enumerating all the segmentations.*

Question Q11 *Give the total error of the optimal APCA for:*

a) $T_0^{(200)}$, $N = 5$ **b)** $T_5^{(500)}$, $N = 10$

c) $T_{10}^{(1000)}$, $N = 10$ **d)** $T_{20}^{(2000)}$, $N = 8$

Question T14 *Is BuAPCA optimal? Give the smallest counter-example you can find, and compare the errors of both algorithms on the series of Question Q11.*

4.4 APCA for similarity search

Let T be represented by $\mathsf{APCA}(T) = \{(x_i, [b_i, e_i])\}_{i<N}$ and let $\bar{q}_i$ be the mean of the query Q over $[b_i, e_i]$. We define

$$\mathsf{LB_APCA}(Q, T) = \sqrt{\textstyle\sum_{i=0}^{N-1}(e_i - b_i + 1)(\bar{q}_i - x_i)^2}.$$

Question T15 *Prove that $\mathsf{LB_APCA}(Q, T) \leq \mathsf{ED}(Q, T)$. What is the complexity of computing it for all the series of the database?*

Question Q12 *Represent each series of D by its bottom-up APCA with N segments. Give the number of series $T \neq NN_{\mathsf{ED}}(Q)$ such that $\mathsf{LB_APCA}(Q, T) < \mathsf{ED}(Q, NN_{\mathsf{ED}}(Q))$ for:*

a) $Q = Q_2$, $N = 4$ **b)** $Q = Q_2$, $N = 8$

c) $Q = Q_3$, $N = 4$ **d)** $Q = Q_3$, $N = 8$

Question T16 (experiment) *APCA with 8 segments uses as much memory as PAA of order 16. Compare the two representations on the database: average reconstruction error, number of candidates (as in Q5 and Q12), tightness* LB/ED. *Does a better approximation of the series always lead to a better pruning? Explain.*

5 Lower bounding DTW and indexing with an R-tree (at home)

In this part, all series have the same length $l = 256$ and we use DTW_w with $w = 10$ unless stated otherwise. Computing DTW_w for every series of the database is expensive; we want a lower bound as tight as possible, and then an index.

5.1 The LB_Keogh lower bound

The *envelope* of a query Q for a band w is given by the two series

$$U_i = \max_{\max(0,i-w)\leq k\leq \min(l-1,i+w)} q_k, \qquad L_i = \min_{\max(0,i-w)\leq k\leq \min(l-1,i+w)} q_k,$$

and, for a candidate series $C = [c_0, \ldots, c_{l-1}]$,

$$\mathsf{LB_Keogh}(Q, C) = \sqrt{\sum_{i=0}^{l-1} \begin{cases} (c_i - U_i)^2 & \text{if } c_i > U_i \\ (c_i - L_i)^2 & \text{if } c_i < L_i \\ 0 & \text{otherwise.} \end{cases}}$$

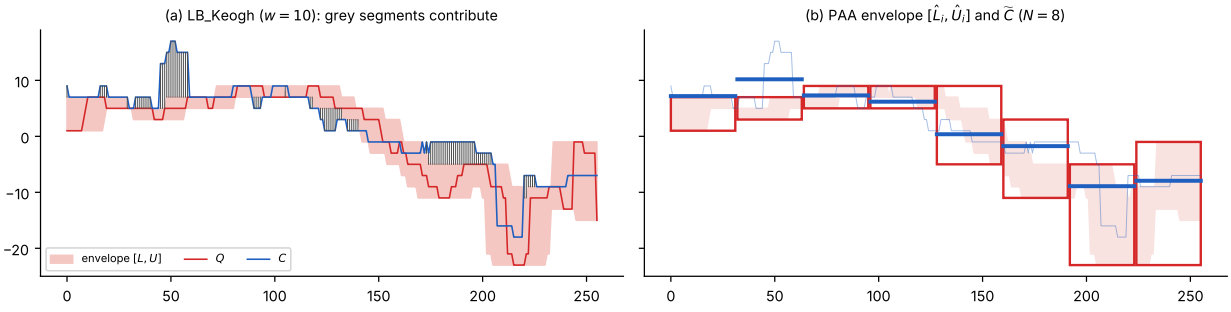


Figure 6: (a) Envelope of Q_3 and a candidate C. (b) The PAA envelope (Section 5.2).

Question T17 *Prove that* $\mathsf{LB_Keogh}(Q, C) \leq \mathsf{DTW}_w(Q, C)$.

Question T18 *What is the complexity of computing the envelope naively? Propose an algorithm computing it in $O(l)$, independently of w.*

Question T19 (experiment) *How does the tightness* $\mathsf{LB_Keogh}/\mathsf{DTW}_w$ *evolve with w? Illustrate on one query and explain.*

Question Q13 *With $w = 10$, give* $\mathsf{LB_Keogh}(Q_j, T_j)$ *for:*

a) $j = 0$ **b)** $j = 1$
c) $j = 2$ **d)** $j = 3$

Question Q14 *Give the number of series $T \neq NN_{\mathsf{DTW}_w}(Q)$ of D such that*

$$\mathsf{LB_Keogh}(Q, T) < \mathsf{DTW}_w(Q, NN_{\mathsf{DTW}_w}(Q))$$

(the envelope and DTW_w *being computed with the same w) for:*

a) $Q = Q_1$, $w = 10$ **b)** $Q = Q_3$, $w = 10$
c) $Q = Q_1$, $w = 20$ **d)** $Q = Q_3$, $w = 20$

5.2 Indexing DTW

LB_Keogh still requires $O(l)$ operations per series. To build an index, we reduce the dimension with the PAA (order N, segments of length l/N). We define the *PAA envelope* of the query, $\hat{U} = [\hat{U}_0, \dots, \hat{U}_{N-1}]$ and $\hat{L} = [\hat{L}_0, \dots, \hat{L}_{N-1}]$, where $\hat{U}_i$ (resp. $\hat{L}_i$) is the **maximum** of U (resp. the **minimum** of L) over the i-th segment, and, with $\widetilde{C} = [\hat{c}_0, \dots, \hat{c}_{N-1}]$ the PAA of C:

$$\mathsf{LB_PAA}(Q, C) = \sqrt{\frac{l}{N} \sum_{i=0}^{N-1} \begin{cases} (\hat{c}_i - \hat{U}_i)^2 & \text{if } \hat{c}_i > \hat{U}_i \\ (\hat{c}_i - \hat{L}_i)^2 & \text{if } \hat{c}_i < \hat{L}_i \\ 0 & \text{otherwise.} \end{cases}}$$

Question T20 *Prove that* $\mathsf{LB_PAA}(Q, C) \leq \mathsf{LB_Keogh}(Q, C)$.

5.3 Background: R-trees

Each series C of the database is now seen as a point $\widetilde{C} \in \mathbb{R}^N$ (its PAA). To avoid scanning all the points, we store them in an *R-tree*, a classical index for multidimensional data.

Minimum bounding rectangle. The MBR of a finite set of points $S \subset \mathbb{R}^N$ is the smallest axis-parallel (hyper-)rectangle containing them:

$$\mathrm{MBR}(S) = \prod_{i=0}^{N-1} [r_i^-, r_i^+], \qquad r_i^- = \min_{p \in S} p_i, \quad r_i^+ = \max_{p \in S} p_i.$$

It is stored with $2N$ numbers.

Structure. An R-tree with parameters $m \leq M/2$ (minimum and maximum number of entries per node) is a rooted tree such that:

- every **leaf** contains between m and M entries (I, id), where id identifies an object and I is the smallest rectangle containing it; here the objects are points, so I is simply the point $p = \widetilde{T}_k$ and $id = k$;
- every **internal node** contains between m and M entries $(R, child)$, where $child$ points to a sub-tree and R is the MBR of the rectangles of the child node, i.e. of *all* the points stored in this sub-tree;
- the root has at least two children (unless it is a leaf), and the bounds m do not apply to it;
- all the leaves are at the same depth (the tree is **balanced**), so that the height is $O(\log_m n)$.

Contrary to a k-d tree, the rectangles of two sibling nodes may **overlap**: a region of the space can be covered by several sub-trees, and a search may have to explore several of them (cf. Figure 7).

Construction. R-trees are usually built by successive insertions (Guttman's algorithm: descend into the child whose MBR needs the smallest enlargement, split a node when it exceeds M entries). When all the points are known in advance, as here, it is simpler and better to *bulk load* (pack) the tree. **The construction is provided** (function `build_rtree`; do not modify it), so that everybody works on the same tree. It implements the *Sort-Tile-Recursive* (STR) packing algorithm with M entries per node:

1. **leaf level.** Let r be the number of points in dimension k and $P = \lceil r/M \rceil$ the number of pages to fill. Sort the points by their first coordinate and cut the sorted list into $S = \lceil P^{1/k} \rceil$ *slabs*, each slab being a run of $M\lceil P^{(k-1)/k} \rceil$ consecutive points (the last slab may be smaller). Each slab is then processed recursively as a $(k-1)$-dimensional data set, using the next coordinate. In dimension 1, the sorted points are cut into runs of M consecutive points (i.e., leaves);

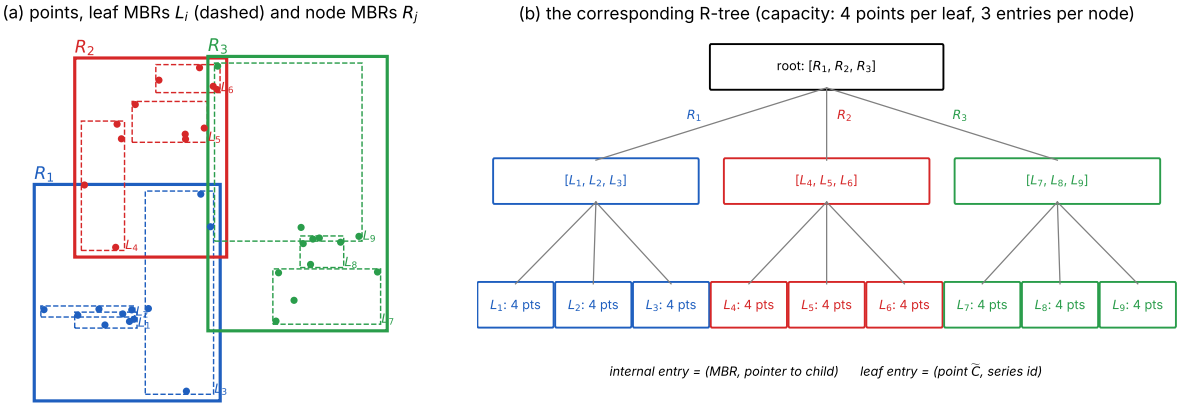


Figure 7: A toy R-tree on 36 points of $\mathbb{R}^2$. Note that the MBRs R_1, R_2, R_3 overlap.

2. **upper levels.** The MBRs of the nodes of a level are packed in the same way, using the centres of the MBRs as coordinates, which produces the nodes of the level above; this is repeated until a single node (the root) remains.

Ties in the sorts are broken by index, and all the coordinates are handled with exact integers, so that the tree is the same on every computer. `build_rtree(D, N, M)` only returns the *structure* of the tree: a leaf is `("leaf", [k_1, ..., k_m])` (indices of series), an internal node is `("node", [child_1, ..., child_m])`. With $n = 1000$, $N = 8$ and $M = 16$, the tree has 63 leaves (each with 8 to 16 series), 4 internal nodes and a root: it satisfies the properties above with $m = 8$. **Computing the MBRs is your job.**

MINDIST. For a point $p \in \mathbb{R}^N$ and a rectangle $R = \prod_i [r_i^-, r_i^+]$, the Euclidean distance from p to R is:

$$\mathsf{MINDIST}(p, R) = \sqrt{\sum_{i=0}^{N-1} g_i^2}, \qquad g_i = \begin{cases} r_i^- - p_i & \text{if } p_i < r_i^- \\ p_i - r_i^+ & \text{if } p_i > r_i^+ \\ 0 & \text{otherwise.} \end{cases}$$

It is computed in $O(N)$ and satisfies two key properties: (i) $\mathsf{MINDIST}(p, R) \leq \|p - x\|$ for every point $x \in R$, hence for every point stored in the sub-tree of R; (ii) if $R' \subseteq R$ (e.g. R' is the MBR of a child of the node of MBR R), then $\mathsf{MINDIST}(p, R') \geq \mathsf{MINDIST}(p, R)$.

Best-first nearest-neighbour search. The standard exact 1-NN search for a point query q explores the tree in increasing order of MINDIST, using a priority queue that contains both nodes and points. When a point is popped, every element remaining in the queue has a key at least as large, and each key is a lower bound of the distance to all the points it contains (property (i)); hence the popped point is the nearest neighbour. Thanks to property (ii), the nodes are visited by increasing MINDIST, and a node is opened only if its MINDIST is smaller than the distance to the nearest neighbour.

5.4 Back to DTW: indexing the PAA

In our setting the query is not a point: it is described by its PAA envelope $(\hat{L}, \hat{U})$, and the distance we are interested in is DTW_w, which is not computed in the PAA space.

Question T21 *Give a geometric interpretation of the PAA envelope $(\hat{L}, \hat{U})$ and of* LB_PAA *in $\mathbb{R}^N$. Adapt the definition of* MINDIST *(Section 5.3) into a function* $\mathsf{MINDIST}(Q, R)$*, computable in $O(N)$ for an MBR R, such that $\mathsf{MINDIST}(Q, R) \leq \mathsf{DTW}_w(Q, C)$ for every series C stored in the sub-tree of R. Prove it, and show that property (ii) still holds.*

Algorithm 1: BestFirstNN$(q, root)$ (Euclidean distance)

```
1 H ← priority queue (min-heap) containing (0, root)
2 while H is not empty do
3     (key, e) ← pop the element of smallest key from H
4     if e is a point (p, id) then
5         return (id, key)                    // no remaining element can be closer
6     else if e is a leaf then
7         foreach entry (p, id) of e do push (‖q − p‖, (p, id)) in H
8     else
9         foreach entry (R, child) of e do push (MINDIST(q, R), child) in H
```

Question T22 *Using the tree returned by* `build_rtree(D, 8)` *($N = 8$, $M = 16$), compute the MBR of every node. Adapt* BestFirstNN *into an exact nearest-neighbour search under* DTW_w*, which uses successively* MINDIST, LB_PAA, LB_Keogh *and finally* DTW_w*, computing the expensive quantities only when necessary. Describe your search algorithm and prove that it returns the exact nearest neighbour. Characterise the set of nodes opened by your search (popped from the priority queue and explored). Show that it can be designed so that the numbers of* LB_Keogh *and of* DTW_w *computations depend only on the lower bounds (and not on the shape of the tree), and relate them to Question Q14.*

Question Q15 *With $w = 10$, $N = 8$ and the tree returned by* `build_rtree(D, 8)`*, give the number of nodes (root, internal nodes and leaves) opened by your best-first search, where* $\mathsf{MINDIST}(Q, R)$ *is the* tightest *lower bound you can derive from an MBR R (i.e. the smallest value of* $\mathsf{LB_PAA}(Q, \cdot)$ *over the points of R). A node is* opened *when it is popped from the priority queue* ***and*** *its entries are examined; nodes still in the queue when the search stops (or popped only to stop the search) are not counted. Give the result for:*

a) $Q = Q_0$ **b)** $Q = Q_1$
c) $Q = Q_2$ **d)** $Q = Q_3$

Question T23 (experiment) *Compare the running times of the brute force, of a sequential scan with* LB_Keogh*, and of your index. Study the influence of N and of the node capacity on the number of visited nodes. Comment.*

◇ ◇ ◇

A Reference answer sheet for $\widetilde{u_0} = 42$

These values are obtained with `SEED = 42`. Use them to test your programs, then run them with your own seed u_0 and report your answers in the dictionary `ANSWERS` of your notebook.

	a)	**b)**	**c)**	**d)**
Q1	27	28	15	22
Q2	181.69	118.22	100.40	240.70
Q3	138.32	165.43	179.14	181.69
Q4	(252, 12.88)	(723, 40.34)	(204, 80.86)	(969, 75.72)
Q5	140	100	50	25
Q6	47.13	83.56	993.47	5869.36
Q7	181.69	161.68	117.60	34104.77
Q8	(723, 37.87)	(194, 14.11)	(969, 62.59)	(8, 63.85)
Q9	(728, 26.88)	(7549, 5316.31)	(86477, 486854.81)	(288523, 874928.55)
Q10	(184, 6025.10)	(1230, 257269.85)	(908, 458615.38)	(2320, 4958095.18)
Q11	673.41	1064.41	6238.07	18741.20
Q12	97	43	57	14
Q13	168.29	70.34	285.16	101.22
Q14	55	25	120	31
Q15	27	47	19	45