APPLICATION-DRIVEN PARTNER LEG PAIRING USING A GEOSPATIAL DATABASE

Kaushikk V. N.	Helmut Schmidt University, Hamburg, Germany
B Bonnett	Helmut Schmidt University, Hamburg, Germany
H Schmaljohann	WTD 71, Eckenförde, Germany
R Klemm	ATLAS ELEKTRONIK GmbH, Bremen, Germany
T Fickenscher	Helmut Schmidt University, Hamburg, Germany

1 INTRODUCTION

Synthetic aperture sonar (SAS) is capable of producing high-resolution images of the seafloor. Many applications of SAS involve comparing data from different missions. For example, image-based change detection can identify areas within the imaged scene where the backscattering strength has changed in the time interval between missions. Similarly, object-based change detection looks for objects which have appeared, disappeared or changed between images. Image fusion can combine multiple images of a scene, typically captured from different aspect angles, to give a single image with advantages other than increasing the image resolution. Applications such as these require finding suitable partner legs from other missions, with each application imposing different selection criteria. In a regularly surveyed area, such as a harbour or important waterway, a large amount of previous data can be available, making manual selection of such partner legs difficult.

This calls for a database with a flexible structure to store relevant information about SAS missions that have been performed. In this paper, we propose a database using the GeoPackage format, a non-proprietary geospatial database widely supported by geographic information system (GIS) platforms and common programming languages. The GeoPackage format defines how geospatial information, such as the trajectory followed by the sonar, should be stored, and allows any desired metadata columns to be added to each table. Further details about both the general GeoPackage database format and the structure of our SAS database are given in Section 2. Retaining every point of the sonar trajectory will be unnecessary for most applications; a method of simplifying trajectories for storage in the database is presented in Section 3.

Two example applications using information from the database are then presented. Both of these employ a hierarchical structure: initial filtering at the database level selects potentially useful missions, and further filtering in the application refines the set of chosen missions. In Section 4, a procedure for finding partner legs for image-based change detection is given. The second application, detailed in Section 5, is to find a set of images of a desired area which can be utilised for some form of multi-image processing. Some discussions and concluding remarks are given in Section 6, and details about the line piece distance metric used in the change detection application are provided in Appendix A.

2 GEOPACKAGE MISSION DATABASE

GeoPackage (sometimes abbreviated to GPKG) is a non-proprietary, platform-independent geospatial database format defined by an Open Geospatial Consortium (OGC) standard.[1] It uses the public domain SQLite database engine which stores data in a single file. As such, it is easy to share and can be directly accessed, i.e., without a separate API or database server, by any programming language with SQLite support (which is widespread). It is widely supported by GIS platforms.

For this work, we use a GeoPackage database with three *feature tables*, that is, tables containing vector features. The geospatial definition of each feature is stored in a *geometry column*; each feature table must have exactly one geometry column. The structure of this database is shown in Figure 1.

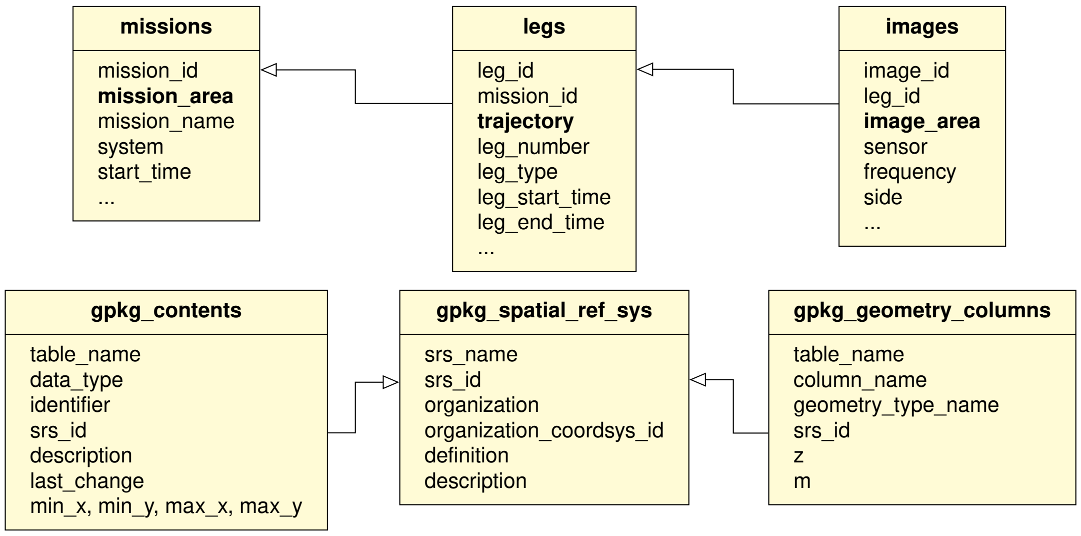


Figure 1: GeoPackage database structure for this work. The upper row shows the feature tables and the lower row the GeoPackage tables. For simplicity, not all metadata columns in the feature tables are listed. The geometry columns are marked in bold.

Information about the complete mission is stored in the `missions` table with a polygon geometry column `mission_area` marking the area imaged during the mission. Each leg of a mission is placed in the `legs` table with a link to its mission using the unique IDs automatically generated by the database. The trajectory followed by the system during the leg is stored in the `trajectory` geometry column as a linestring (a sequence of points). In general, multiple images can be reconstructed from each leg, one for each combination of receiver, operating frequency etc. Each possible image is stored in the `images` table with the `image_area` geometry column storing a polygon marking the area where an image can be formed. Again, this is linked to the leg via its unique feature ID. All of these tables contain extra metadata columns about the data collection. These columns are not restricted by the GeoPackage standard and so can be trivially customised to suit the system, operator, intended application etc.

Two GeoPackage informational tables are always required. The `gpkg_spatial_ref_sys` table stores each spatial reference system (SRS; a synonym of coordinate reference system, CRS) used within the database using the well-known text (WKT) format[2]. Information about all data tables is stored in the `gpkg_contents` table, including the type of data they store, the SRS in use, a human-readable identifier and description, the timestamp the table was last modified, and a bounding box of the features in the table. As this database contains feature tables, the `gpkg_geometry_columns` is also required. This specifies the type of geometry they use, the SRS and whether the features include z (height) and m (other measure, e.g., linear distance from the start of the feature for each point in a linestring) information. GeoPackages can also store raster images in tile pyramids, and extensions can be specified within a `gpkg_extensions` table. OGC registered extensions include table relationships (see Section 6 for some discussion of this extension) and spatial indexes using R-trees for faster data selection.

3 SIMPLIFYING TRAJECTORIES FOR STORAGE

Most navigation systems will record the position of the sonar at a relatively high rate. For example, the inertial navigation system (INS) on one of our platforms records on the order of 12 000 positions per kilometre travelled in its standard configuration. Although the entire trajectory could be stored within the database, a decimated version will reduce both the database size and the time required to load, process and display the data. The minimal case is to store only the first and last points in the database, but for some applications it may be desirable to store more points. In the following, we propose a method to simplify a trajectory to N points denoted $p_1, p_2, \ldots, p_N$ forming $N - 1$ line segments.

Many line simplification techniques only consider the error between the original and simplified lines. We wish to impose three additional constraints on the segments of the simplified trajectory. Firstly, each of the N points must correspond to points in the original trajectory. Similarly, p_1 and p_N must be the first and last points of the original trajectory, respectively. Finally, the length L_n of the n^{th} segment (that is, the segment between p_n and p_{n+1}) should be within some range of values $[L_{\text{min}}, L_{\text{max}}]$. Here, we take L_n as the cumulative linear distance travelled by the sonar between p_n and p_{n+1} on the original trajectory; similar results could be achieved by using the direct distance $|p_{n+1} - p_n|$. The following iterative algorithm performs the simplification using a parameter $0 < \alpha < 1$ to reduce the size of each tested segment in a geometric progression until it is suitable:

1. Initialise $n = 1$ and set p_1 to the first point of the original trajectory.
2. Set the proposed length $\tilde{L}_n = L_{\text{max}}$.
3. Select all points in the original trajectory within distance $\tilde{L}_n$ from p_n. The furthest of these points from p_n is the proposed segment end point $\tilde{p}_{n+1}$.
4. Compute some error metric ϵ between the selected points and the line between p_n and $\tilde{p}_{n+1}$. If ϵ fails to meet a predetermined threshold ϵ_T, decrease the proposed length to $\tilde{L}_n = \alpha\tilde{L}_n$ and if the reduced $\tilde{L}_n > L_{\text{min}}$ return to step 3.
5. As either the error or minimum length criteria have been met, we have found p_{n+1}. If p_{n+1} is closer than distance L_{min} to the end of the original trajectory, set p_{n+2} to that end point and stop; we have now completed simplification of the trajectory to $N = n + 2$ points.
6. Increment n and return to step 2 to process the next segment.

Note that the final segment may be shorter than L_{min} with this procedure. If preferred, step 5 may be modified to replace p_{n+1} with the original end point in the stop condition. A maximum of

$$M_{\text{max}} = \left\lceil \frac{\log(L_{\text{min}}/L_{\text{max}})}{\log \alpha} \right\rceil + 1 \tag{1}$$

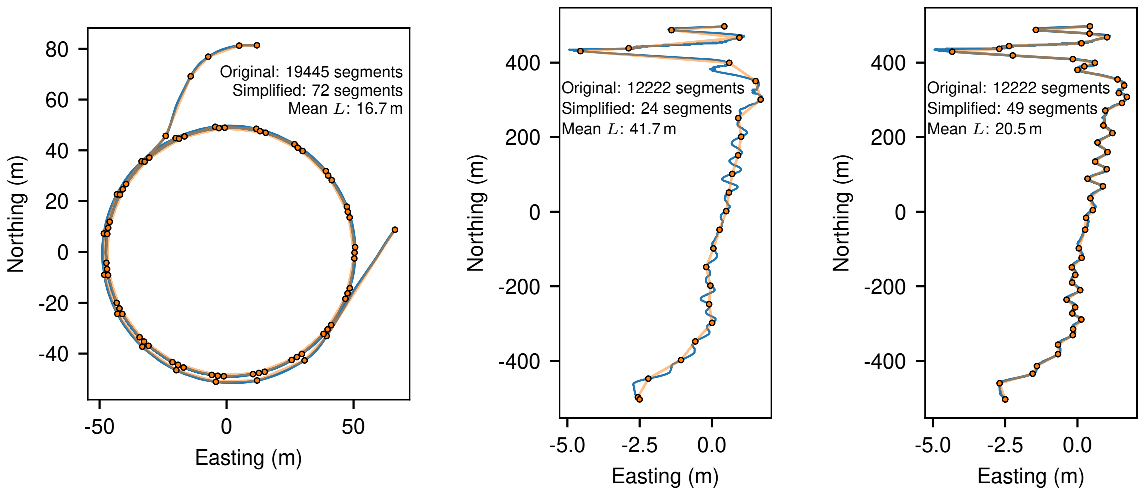
lengths will be tested per segment, giving an upper bound of $\lceil M_{\text{max}} L_{\text{full}}/L_{\text{min}} \rceil$ tests to simplify a trajectory of length L_{full}.

The simplification of some recorded trajectories is shown in Figure 2 using the mean of the perpendicular distances between the original points and the proposed line segment for the metric ϵ with $\alpha = 0.8$, $L_{\text{min}} = 10$ m and $L_{\text{max}} = 50$ m. These length parameters were empirically found to yield good results with a variety of trajectories recorded by our system. Comparing Figures 2b and 2c shows that the error threshold can be chosen to determine whether the algorithm follows wobbles in the trajectory with shorter segments or whether it uses longer segments in the mean direction of travel.

4 FINDING PARTNER LEGS FOR CHANGE DETECTION

Change detection is a common application of SAS images. Ideally, the only differences between two images would be due to changes in the imaged scene. This requires other sources of differences, such as a change in imaging geometry, to be minimised. As registering the images and performing change detection is a complex procedure, it is desirable to identify potential partner legs from a comparison of their trajectories. Given a mission database as described previously and a base leg from that database, we propose the following procedure to determine other legs which have a high likelihood of being suitable for change detection against the base leg:

1. Select all other legs which intersect the boundary polygon of the base mission as the initial set of candidate legs.
2. If the database uses geographic coordinates, project all trajectories to Cartesian coordinates.



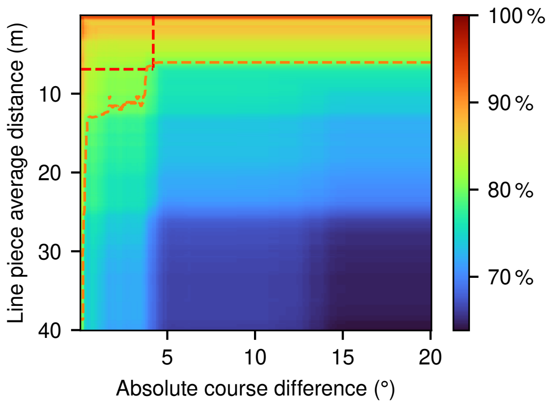
(a) Four circles with $\epsilon_T = 0.5\,\mathrm{m}$ *(b)* $\epsilon_T = 0.5\,\mathrm{m}$ *(c)* $\epsilon_T = 5\,\mathrm{mm}$

Figure 2: Examples of trajectory simplification. Blue lines denote the original trajectory and orange lines the simplified version. Note the exaggerated horizontal axes in (b) and (c).

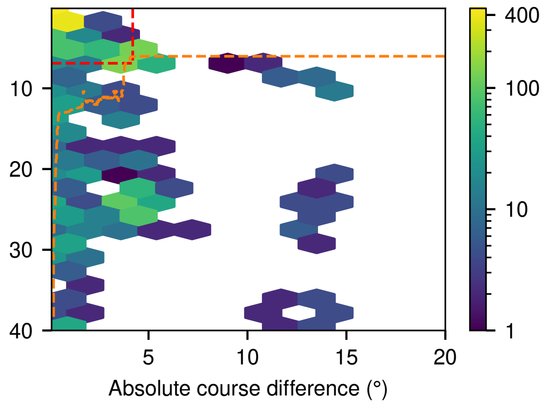
3. Remove legs which do not meet predetermined thresholds, e.g., are not within a certain distance of the base leg or differ in heading by more than an allowable amount, from the candidate set.
4. Estimate the change detection success rate for the base leg and each candidate leg, and select those above a set threshold as the partner legs.

To demonstrate this algorithm, an implementation was created in Python using the fudgeo package to interact with the database, the SpatiaLite[3] extension library to efficiently perform the intersection as part of the SQL `SELECT` query, and NumPy to perform the remaining calculations. Two metrics were used for the third step: the line piece average distance (see Appendix A) and the absolute course difference between the legs. These metrics, amongst others, were previously analysed by Klemm *et al.*[4] as potential predictors of change detection performance. In that work, upper limits of 6.9 m for the line piece average distance and 4.2° for the course difference were suggested. The success rate of change detection over their dataset (defined as the percentage of leg pairs meeting a given distance and course limit which could be successfully registered for change detection) is reproduced in Figure 3a. This suggests a high likelihood of successful change detection for combinations of parameters outside one of these limits; note that caution is required when considering the results for higher course differences since, as Figure 3b shows, few trials are available in this region. To take advantage of this, the threshold for the line piece average distance was increased to 14 m and the provided success rate used for the final step with a lower bound of 80 % required.

To evaluate the performance of this algorithm, four databases containing randomly generated mission, leg and image data were created. These differed in size (1980 legs and 18 383 legs) and the coordinate system used for geospatial data (geographic coordinates with the EPSG:4326 or GPS coordinate system, and Cartesian coordinates in the EPSG:32632 or UTM zone 32N coordinate system). A single-threaded Python implementation of the algorithm was written, and 10 000 trials of this was run over each of the databases on an Intel i9-12900K CPU. Each trial used a different randomly selected base leg from the database. Figure 4 shows the mean and standard deviation of the time taken to complete the algorithm as a function of the number of potential partner legs found. The size of the database has the largest impact on the performance, with Cartesian coordinates being faster than geographic coordinates for databases of the same size. There appears to be no significant link between the processing time and the number of partners found.



(a) Success rate.



(b) Number of trials in each region of the parameter space.

Figure 3: (a) The success rate of the change detection registration and (b) the parameter space distribution of trials used to generate (a). The red dashed lines indicate the 6.9 m and 4.2° thresholds and the orange dashed line shows the 80 % success contour.

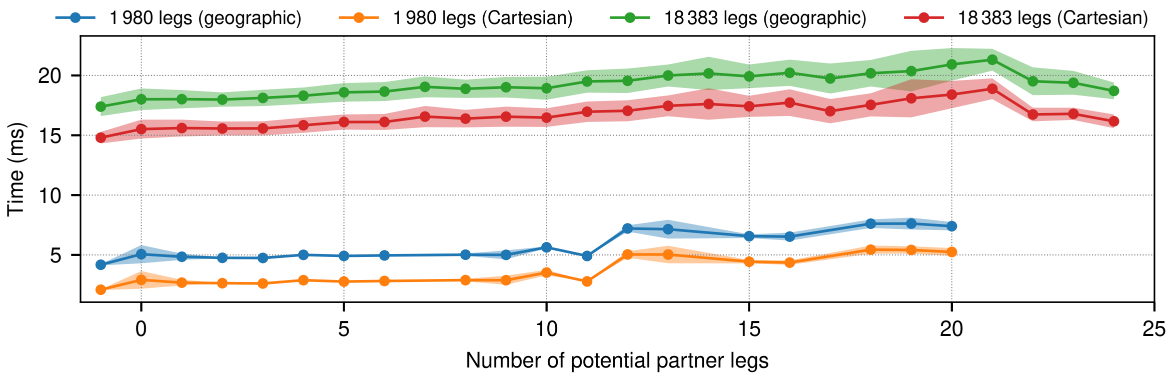


Figure 4: Mean (solid line) and standard deviation (shaded area) of the time taken to find potential change detection partner legs from different databases over 10 000 trials. -1 on the horizontal axis corresponds to no initial matches from the database.

5 FINDING PARTNER LEGS FOR MULTI-IMAGE PROCESSING

Multiple SAS images of a common area can be combined to improve analysis. For example, images of an area collected on multiple legs with different aspect angles can be combined to reduce shadows and speckle noise[5] or used to improve the performance of an object classifier.[6] For some applications, images from different systems, for example with different resolutions and clutter suppression abilities, can be combined[7] while other applications will require the same platform to be used for all images. Starting from an area of interest (AOI), such as an existing image or a polygon around a detected object, we propose the following algorithm for finding partner images:

1. Select all images in the database which intersect the AOI as the initial candidate set. Other criteria based on stored data, e.g., mission date or system used, can be applied here.
2. If the database uses geographic coordinates, project all trajectories and polygons to Cartesian coordinates.
3. Measure the area of the intersection polygon for each candidate and discard any candidate for which this area is too low.

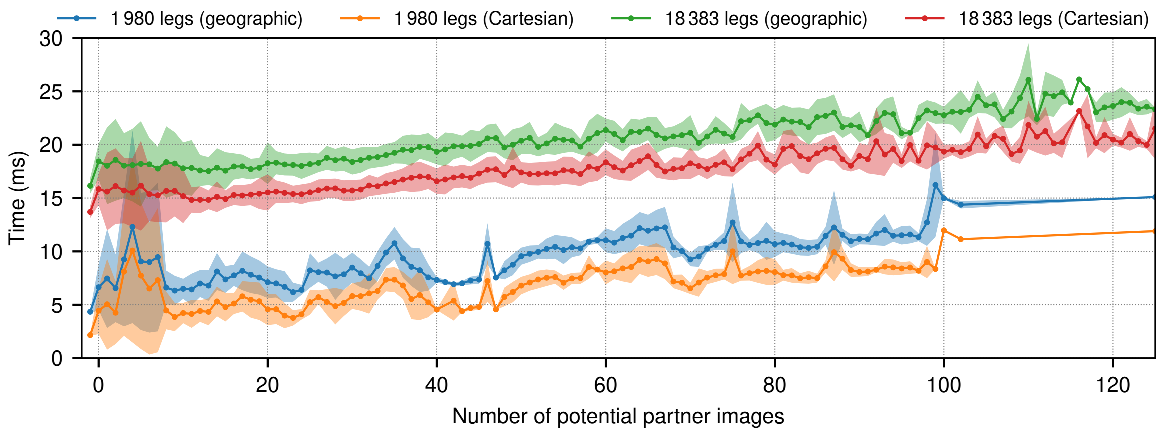


Figure 5: Mean (solid line) and standard deviation (shaded area) of the time taken to find potential partner images from databases containing different numbers of legs and using different coordinate systems. 10 000 trials were performed, each using a different base image randomly selected from the database. -1 on the horizontal axis corresponds to no initial matches from the database.

4. Determine a view reference point. The centre of the AOI is one option. For a large image with intersections in widely separated regions, multiple reference points could be used, for example by clustering the intersections and using the centroid of each cluster.
5. Find the view vectors from the reference point to each vertex in the candidate trajectories. Calculate the view angle and view range as the angle from east of the vector and the length of the vector, respectively. Select any candidates which meet desired criteria on the view angle and range as the partners.

The criteria in the final step can be complex. For example, it may be sufficient to use the median view angle, or more detailed analysis of the sequence of view angles may be required. Multiple images from similar view angles and ranges may be acceptable, or an extra selection step may be required, e.g., out of a set of images from similar view angles and ranges, select the closest in time to the base image.

An single-threaded implementation of this algorithm was coded in Python using the fudgeo package to interact with the database, the SpatiaLite extension to implement the intersection test and provide a polygon of the intersection, and NumPy to perform the remaining calculations. It was tested on the same set of four databases used in the previous section. For each database, 10 000 trials were performed on an Intel i9-12900K processor, each using a different randomly chosen base image and a fixed minimum intersection area of 10 000 m^2. The mean and standard deviation of the time taken to complete the algorithm as a function of the number of image partners found is shown in Figure 5. As with the change detection application, the size of the database has a large impact, and Cartesian coordinates are faster. A gradual increase in processing time can be observed as the number of partner images found increases.

6 DISCUSSION

GeoPackage is a flexible database format widely supported by many GIS systems and common programming language. As demonstrated by the examples applications presented here, it is suitable for storing and filtering SAS mission information. Its ability to store images in raster tile layers has not been tested in this work, but no obvious reason is seen why a GeoPackage database could not be used for storing SAS images. It should be noted that such tiles are encoded as PNG or JPEG images, and so would only be suitable for display rather than subsequent complex analysis. Being based on

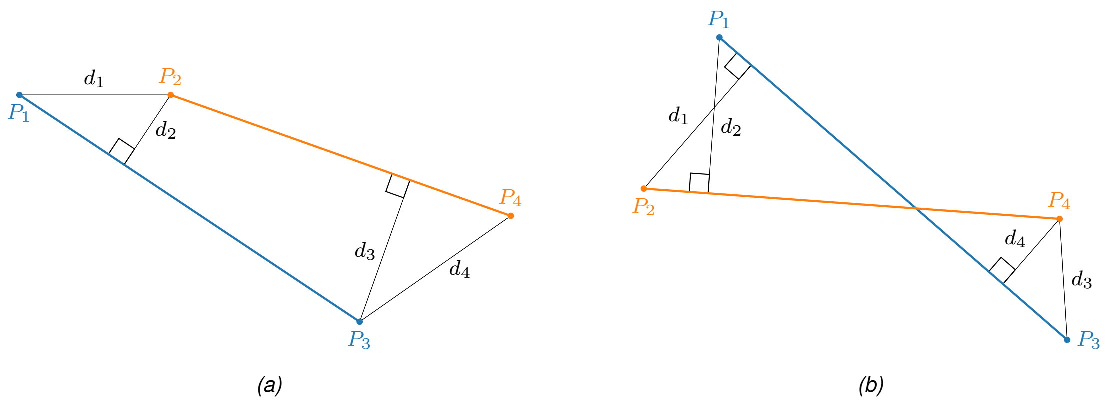


Figure 6: Two examples of the intermediate distances used to calculate the line piece distances. The base leg is drawn in blue and the partner leg in orange.

SQLite means a database is easy to share, and its performance for the example applications was no issue. More high-performance databases for geospatial applications, such as PostgreSQL+PostGIS, are available but are more complicated to run and are not easily portable.

The database structure used here assumes a link between entries in the `images` and `legs` tables, and between entries in the `legs` and `missions` tables. In SQL terms, this can be described with a foreign key within the database. SQLite has support for foreign keys, but this must be enabled at compile time by each application using the SQLite engine. Even if it is, foreign keys are not enforced by default, with a special `PRAGMA` query required at the start of each connection to enable enforcement. An external application which does not enforce this may therefore modify the database to an inconsistent state. The GeoPackage related tables extension[8] adds support for defining relationships between data tables by adding a mapping table for each link. These mapping tables can contain extra metadata about the relationship, and can therefore be more flexible than a foreign key. An external application which does not support this extension may modify the data tables without updating the mapping, again resulting in an inconsistent database. Software reading from a database should therefore be prepared to handle these situations.

A GeoPackage database can store information with both geographic SRSs and projected SRSs. Most functions in the SpatiaLite extension library can also work with either type of SRS. In the applications presented here, using a projected SRS had a minor performance improvement. Note that projections have a usable area outside of which the accuracy of the projection degrades. In partner applications such as presented here, a similar level of error will apply to all nearby data, and so the error in the differences may not be significant.

Some example code for generating a database, implementing the example algorithms and reproducing the timing results is available at `https://github.com/hsu-sonar/icua24-geopackage` under the CC0 license (equivalent to public domain in most jurisdictions).

APPENDIX A

Two legs (or pieces of legs) being compared can be represented as two closed line segments. Label the starting and finishing vertices of the base leg as P_1 and P_3, respectively. There are two options for the labels P_2 and P_4, the vertices of the partner leg. Compute $u = \min(||P_2 - P_1||\,, ||P_4 - P_3||)$ for each option, and select the option which minimises u. For each point P_n, find a point H_n on the other leg such that the line segment $\overline{P_n H_n}$ is perpendicular to the other leg. If no such point exists within the other leg, set H_n to be the endpoint of the other leg closest to perpendicular. The distance d_n is then the Euclidean distance between P_n and H_n. Figure 6 shows some examples of these distances.

The *line piece minimum distance* d_{min} represents the closest approach between the two legs:

$$d_{\text{min}} = \begin{cases} 0 & \text{if } \overline{P_1P_3} \text{ and } \overline{P_2P_4} \text{ intersect,} \\ \min\limits_{n \in 1,\ldots,4} d_n & \text{otherwise.} \end{cases} \tag{2}$$

The *line piece average distance* $\bar{d}$ is the mean of the smaller of the distances at either end of the segments, adjusted for intersecting trajectories:

$$\bar{d} = \begin{cases} \dfrac{1}{4}\left(\min\limits_{n \in 1,2} d_n + \min\limits_{m \in 3,4} d_m\right) & \text{if } \overline{P_1P_3} \text{ and } \overline{P_2P_4} \text{ intersect,} \\ \dfrac{1}{2}\left(\min\limits_{n \in 1,2} d_n + \min\limits_{m \in 3,4} d_m\right) & \text{otherwise.} \end{cases} \tag{3}$$

REFERENCES

1. Open Geospatial Consortium, "GeoPackage encoding standard," version 1.4.0 (February 6, 2024). Available: `http://www.opengis.net/doc/IS/geopackage/1.4`.
2. Open Geospatial Consortium, "OpenGIS® implementation specification: Coordinate transformation services," version 1.0.0 (January 12, 2001). Available: `https://www.ogc.org/standard/ct/`.
3. A. Furieri and contributors, *SpatiaLite*. Available: `https://www.gaia-gis.it/fossil/libspatialite/`.
4. R. Klemm, J. Groen, and H. Schmaljohann, "Interoperable image-based change detection," *Proceedings of the 5th International Conference on Synthetic Aperture in Sonar and Radar*, Lerici (September 2023).
5. J. Dillon and S.-M. Steele, "Square SAS: Multi-aspect imaging with a towed synthetic aperture sonar," *OCEANS 2022 - Chennai*, 1–5 (2022).
6. D. P. Williams and S. Dugelay, "Multi-view SAS image classification using deep learning," *OCEANS 2016 MTS/IEEE Monterey*, 1–9 (2016).
7. J. D. Tucker and M. R. Azimi-Sadjadi, "Coherence-based underwater target detection from multiple disparate sonar platforms," *IEEE Journal of Oceanic Engineering* **36**(1) 37–51 (2011).
8. Open Geospatial Consortium, "OGC GeoPackage related tables extension," version 1.0 (May 8, 2019). Available: `http://www.opengis.net/doc/IS/gpkg-rte/1.0`.