Detection of Active Deformation Areas based on Sentinel-1 imagery: an efficient, fast and flexible implementation

José A. Navarro, María Cuevas-González, Anna Barra, Michele Crosetto
Centre Tecnològic de Telecomunicacions de Catalunya (CTTC/CERCA), Av. Carl Friedrich Gauss 7, 08860 Castelldefels, Spain
(jose.navarro, maria.cuevas, anna.barra, michele.crosetto)@cttc.es

1. Introduction

The MOMIT project (see [4] for details) aims at developing and demonstrating a new use of remote sensing technologies for railway infrastructures monitoring. MOMIT solutions will mainly aim at supporting the maintenance and prevention processes within the infrastructure management lifecycle. The overall concept underpinning MOMIT project is the demonstration of the benefits brought by Earth Observation and Remote Sensing to the monitoring of railways networks both in terms of the infrastructure and of the surrounding environment, where activities and phenomena impacting the infrastructure could be present. MOMIT will leverage on state of the art technologies in the fields of space-based remote sensing and RPAS (Remotely Piloted Aircraft Systems) based to perform different kind of analysis thanks to the wide variety of sensors they could be equipped with.

To achieve its goals, six demonstrators showing how these data and technologies may contribute to such objectives are being built, namely:

1. Ground movements nearby the infrastructure.
2. Hydraulic activities nearby the track.
3. Global supervision for natural hazards.
4. Electrical system monitoring.
5. Civil engineering structures monitoring.
6. Safety monitoring.

The Division of Geomatics of the CTTC (Centre Tecnològic de Telecomunicacions de Catalunya) takes care of building some of the components integrating the first demonstrator, whose objectives are detailed in [4]. The goal most relevant to the work presented here is to use the Persistent Scatterer Interferometry method for monitoring the sites of interest, allowing the measurement of differential deformation of the ground and each single structure, with millimetric / centimetric precision.

Several software applications are needed in the context of this demonstrator. This paper describes the first of these, the ADAFinder, a tool to detect and update ADAs (Active Deformation Areas) using Sentinel-1 imagery and the Persistent Scatterer Interferometry technique. The main goal of such application is to update and assess the geohazard activity (volcanic activity, landslides or ground subsidence among other phenomena) of a given area.

A procedure to perform the identification and assessment of ADAs was presented in [1]. In that work, the authors explain in detail the procedure to identify the active deformation areas and also to assess the certainty of such findings; the set of points making each ADA as well as its area of influence and the quality index stating the goodness of the assessment are the most relevant outputs. The input, the set of PSs (Persistent Scatterers, *aka* the points) covering the area to analyze.

The aforementioned procedure relies heavily on the use of a GIS (Geographic Information System) tool and the expertise of its operator; therefore, it is a time-consuming, error-prone process, which requires qualified human resources.

This paper presents an implementation of the method described in [1], the ADAFinder

application. Its goals are (1) to automate the procedure to avoid unnecessary human errors, (2) to reduce the time needed to identify de ADAs, thus opening the door to more frequent updates, and (3) to reduce the expertise required to obtain such results, therefore requiring less qualified people or no human intervention at all, being possible to integrate the process in an automated production workflow, if necessary.

Figure 1 shows a few ADAs and their assessed quality - shown as green (best quality), blue (good), yellow (not so good) and red (bad) areas.

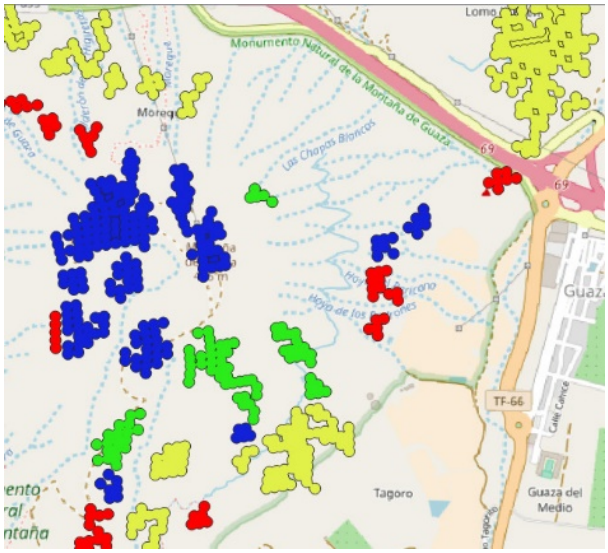

Figure 1: example of a few ADAs.

2. The procedure in a nutshell. Why clustering is so important

The procedure to identify and assess the ADAs is exhaustively described in [1], so it will only be sketched here for the sake of clarity and completeness. The parts of the method that deserved special attention in the implementation of the ADAFinder will be, on the contrary, highlighted.

Figure 2 depicts the method, which is divided into three main steps, namely filtering, ADA extraction and quality assessment. The second and third steps are mainly related to number crunching – qualitatively, not quantitatively speaking – and classification. The mathematics involved there imply the computation of a few correlations, autocorrelations, minimum, maximum and mean values among other miscellaneous intermediate results; although the mathematical part of an algorithm may always be somehow optimized for better performance, such improvements are not the target of this paper. Note, however, that the ADAFinder *does* optimize all the mathematical operations.

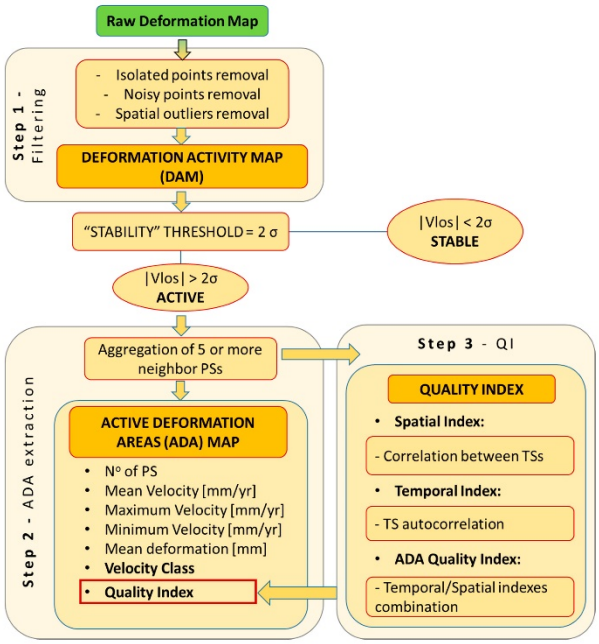

Figure 2: flowchart depicting the algorithm to generate the Active Deformation Areas (**source**: [1]).

Instead, the way to find *groups of points* is much more interesting and challenging, becoming the path leading to a fast implementation of the ADAFinder. Thus, this paper concentrates specifically in the clustering solution implemented by this application.

The filtering step of the algorithm in Figure 2 takes care of removing *isolated* points. In the context of the procedure, isolated stands for points with zero or one neighbors – or, in other words, finding groups including only one or two points. Step 2, the ADA extraction, also expects to find groups, in this case containing five or more points (neighbors), to afterwards check whether these become ADAs or not.

To decide if two points are neighbors, that is, whether they are part of the same group because they are close enough, distances must be computed and then compared to a discriminating threshold. Problems immediately show up when the number of points to check increases. A naïve approach to solving this problem would take all the possible couples of points and compute their respective distances, thus leading to an $O(n^2)$ implementation. Since the number of points usually found in this kind of problems may easily reach hundreds of thousands if not millions, the simple approach just described is simply unacceptable.

3. The clustering algorithm. The point set seen as a graph

The approach adopted by the ADAFinder to find groups (or clusters) of points is to interpret these as a *graph* and then use a classic algorithm, relying on a DFS (Depth First Search), to identify its *connected components*. See, for instance, [3] for further details on this algorithm.

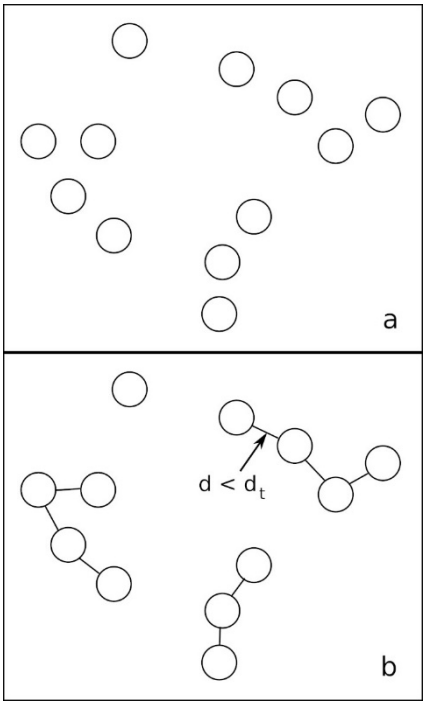

Figure 3: the point dataset (a) seen as a graph with some connected components (b) linked with "distance" arcs.

However, the input to the ADAFinder is a set of points that are in no way connected; that is, data are not organized as a graph (see Figure 3a).

The idea – which makes possible to use the algorithm to find connected components – is to interpret distances as candidate arcs: two points are connected, and therefore an arc exist between these, if and only if the distance d between the points is less than some threshold d_t (see Figure 3b). Having thus a graph, the connected components found by the DFS algorithm are groups of points that are close enough to become clusters – which is the goal sought

The problem here is that the *distance arcs* are not present in the dataset and a process to determine on-the-fly whether these exist is required, which, as stated in section 2, may be a costly operation if addressed in a naïve way.

The solution adopted to reduce the computational burden of the clustering algorithm consists, essentially, in sorting the input data, the points, appropriately: two values are used as sort criteria; first, the x coordinate, then, the y coordinate. The way the array of points is rearranged allows for a minimum set of operations and comparisons.

For instance, the algorithm to find a single connected component takes point i as the first candidate to be included. Its x coordinate, x_i, is then compared to that of the next point in the array, x_{i+1}. If the distance d – obtained just *subtracting* x_i from x_{i+1} – is less than the threshold distance d_t, then, and only then, the full Euclidean distance between points i and $i+1$ is computed and checked once more for the threshold condition. Points satisfying the distance criterion are marked as belonging to the current connected component and stacked for further processing (so its neighbors may be included too in the connected component if these also match the distance criterion). On the contrary, those points too far apart ($d > d_t$) are left aside for later inclusion in other potential connected components.

The key for reducing drastically the number of operation lies in the fact that if the first comparison between x_i and x_{i+1} only – not the full Euclidean distance comparison – fails because these x coordinates are already too far apart, it is not necessary to continue the search for additional points close to reference point *i*. Since the points have been sorted first by x, then by y, it is possible to guarantee that if the distance d_{ij} between points *i* and *j* is greater than the threshold, then d_{ik}, for all $k > j$ will also exceed such limit.

Figure 4 shows a *recursive* version of the connected components finder algorithm just described using pseudocode. The actual implementation avoids recursion and resorts to an iterative version using a stack for performance reasons.

```
PROCEDURE find_component (start, last, cc)
  # start and last and the first and last
  # positions to search in the array of
  # points. cc is a connected component.
  x1 = x_coordinate[start]
  y1 = y_coordinate[start]
  FOR (index between (start, end))
    x2 = x_coordinate[index]
    IF (x2 - x1 > threshold) FINISH
    y2 = y_coordinate[index]
    d = distance(x1,y1,x2,y2)
    IF (d < threshold)
      flag index as visited
      cc.insert(index)
      find_component(index+1, last, cc)
    END_IF
  END_FOR
  FINISH
END_PROCEDURE
```

Figure 4: recursive version of the algorithm finding a single connected component using "distance" arcs between points.

Since the goal of the clustering algorithm is to find *all* the connected components in the dataset, the procedure described above is invoked repeatedly until all the points have been flagged as visited – thus being part of one or another component, even 1-element clusters.

4. Implementation and performance

The ADAFinder application has been *implemented in C++* to boost performance. Other popular languages, as Python, have been avoided precisely for that reason.

Although developed using Microsoft's Visual Studio, special precautions have been taken to *make the source code portable*, particularly for the most popular C++ compiler used in the Linux operating system, i.e. gcc.

In order to ease the integration of the application in a regular production – or research – workflow, the relevant inputs and outputs of the application are ESRI *shapefiles* (see [2] for a formal description of this popular format). Shapefiles are accepted as a regular input / output mechanism by many GIS environments. Among these, ArcGIS or Quantum GIS – *aka* QGIS – are notorious examples. Therefore, using shapefiles to store the relevant data eliminates the impedance that other formats might introduce.

The files using the shapefile format include the input points (PSs) file, an optional polygon file to define the limits of one or more areas of interest to be processed and, finally the output file including the ADAs identified by the application.

Working with shapefiles in C++ is solved by means of the "Shapelib C library" (see [6]).

However, shapefiles are files that may adopt many shapes. The main input of the application, the file containing the points to process, may have been originated in many different ways; therefore, the actual contents may vary from one source to other.

This is especially true in the case of the *dbf* file, where the *attributes* of the several entities present in the shapefile itself are stored. The set of these attributes may vary quite a lot, depending on the application that created the shapefile.

This variability might become a serious problem for ADAFinder, since the input

module should be adapted for each kind of shapefile that should be processed with this tool. However, the use of *read-map files* avoids this problem.

There is a *minimum set of attributes* that must be present in the input dbf file for the ADAFinder to work properly. These are, for each point in the file, the x & y (projected) coordinates, its velocity and the time series stating the different movements undergone by the point. Any other attributes that might be present in the attributes file are simply ignored.

Providing that a dbf file includes the aforementioned attributes, knowing their (column) positions makes possible a general read procedure in ADAFinder, thus avoiding the problem of format variability. Such positions are provided by means of a simple file, namely *the read map file*, including label / value pairs defining the positions of the required attributes (see Figure 5 for an actual example).

```
POSITION_X            =  5
POSITION_Y            =  6
POSITION_VELOCITY     =  9
POSITION_TIME_SERIES  = 11
N_VALUES_TIME_SERIES  = 50
```

Figure 5: an example of a read-map file, showing where to find the different mandatory attributes in some dbf file.

Usually, read map files are valid for whole projects, where the format of the input shapefiles (and related attribute, dbf files) do not vary. At any rate, changing a simple plain text file is enough to adapt the application very quickly – providing that the mandatory attributes exist.

The results of the procedure described in [1] and sketched in Figure 2 depend on the values of a set of parameters; for instance, the stability threshold is 2σ; or the minimum number of points in a cluster to consider it as a candidate ADA is 5. These values (2 and 5 in this case), that might be perfect for some data set, might need some fine-tuning when other data are processed.

Therefore, hardwiring these values in the source code of the application would imply the need to modify it in case they had to be changed.

The ADAFinder identifies all the values that an operator might wish to change in search of better results and lets him / her modify these via *an external options file*.

Again, as in the case of the read map files just described, option files are very simple, consisting of a series of label / value couples in a plain text file.

The ADAFinder is available in three *flavors:*

- A *library*, the ADAFinderLib, ready to be used by any other software components, thus *embedding* the capability of identifying ADAs in other applications.
- A command-line executable of ADAFinder, ready to be integrated in offline (batch) workflows.
- A GUI (Graphic User Interface) version, aimed at interactive use. See Figure 6 for a graphical depiction of the main screen of the application. For portability reasons, the GUI has been implemented using the Qt framework (see [5]).

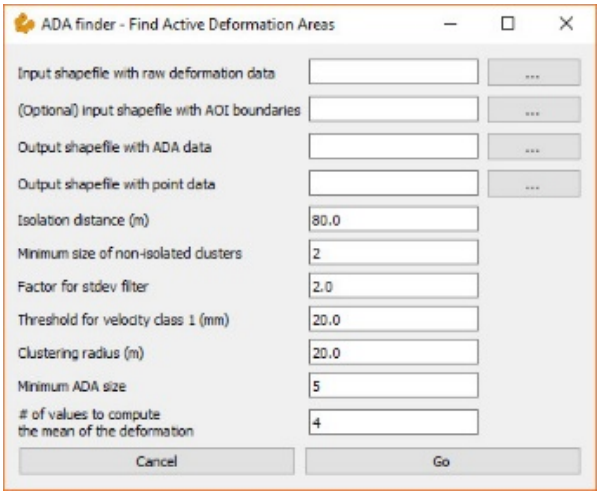

Figure 6: the interface of the GUI-based version of the ADAFinder.

Note that both the command-line and GUI-based versions of the applications rely on the ADAFinderLib library. In fact, these are just a way to collect the information required by the library to work, since it is in the library where the actual logic resides.

The command-line tool collects such information directly from the option and read map files just described. The GUI version reads a "seed" options file to offer its values as defaults in its main screen (see again Figure 6) so the operator may change these at his / her will.

Concerning *performance*, the application needs just a bare second to produce the output ADA map when the number of points (PSs) to process is below 50.000. This makes the ADAFinder a truly interactive tool. Load tests addressed to check the ability of the application to deal with much higher number of points were also conducted. An area covered by almost a million of points was processed; the tool needed less than 3 minutes to produce the output ADAs. It is difficult to state how long an operator would have needed to complete the same work since it depends on his / her level of expertise and the power of the computer used, but it is reasonable to assume that 30 minutes would be a close guess.

It is worth to note that the equipment used to obtain such results is a regular laptop sporting 8 Gb of RAM (Random Access Memory), a rather old, 5th generation Core i5 processor and a classic magnetic disk (that is, not a much faster SSD, Solid State Disk). No GPU (Graphic Processing Unit) processing nor parallelization techniques have been used, so there is still room for improvement – if needed. Consequently, any computer able to run GIS (Geographic Information System) software – a usual environment used for this kind of applications – may be used to run the ADAFinder (either the command-line or GUI versions).

5. Conclusions and outlook

ADAFinder is the first tool that will be part of the first demonstrator ("Ground movements nearby the infrastructure") of the MOMIT project.

This tool, in either of its three incarnations (library, command line, GUI) automates the methodology presented in [1], empowering its users with the ability to perform ADA detection and assessment in very short times, freeing them from the prone-error, manual process required should a GIS environment be used instead. The clustering algorithm implemented makes possible a comparatively very good performance boost (at least 10 times faster than manual operation).

The tool is also flexible, being able to adapt itself to different data formats just changing the file defining how the input point's attributes are structured. This, together with the ability to change the set of parameters controlling the behavior of the application – the options file or the GUI in the interactive version – makes the ADAFinder a very suitable tool to check how different scenarios would change the results; thus increasing the analysis capacity of the operator.

The ADAFinder may be easily integrated in either ArcGIS or Quantum GIS – there is no need to modify the source code; this may be done using the mechanisms provided by these GIS tools. In this way, the ADAFinder may be seen as one more tool in the regular work environment of the operator, avoiding unnecessary disruptions in his / her workflow.

Other tools will follow in the context of the MOMIT project and its firs demonstrator. The next one, already being implemented is the *ADAClassifier*, a tool to identify the kind of ground movement the ADAs are exposing (as, for instance, sinkholes, subsidence, landslides, thermal effects or constructive settlements).

Acknowledgements

The MOMIT project has received funding from the Shift2Rail Joint Undertaking under the European Union's Horizon 2020 research and innovation programme (grant agreement number 777630).

References

1. Barra, A.; Solari, L.; Béjar-Pizarro, M.; Monserrat, O.; Bianchini, S.; Herrera, G.; Crosetto, M.; Sarro, R.; González-Alonso, E.; Mateos, R. M.; Ligüerzana, S.; López, C.; Moretti, S. 2017. "A methodology to detect and update active deformation areas based on Sentinel-1 SAR images." *Remote Sensing*, 9, no. 10: 1002.
2. ESRI, 1998. ESRI shapefile technical description – an ESRI whitepaper (http://downloads.esri.com/support/whitepapers/mo_/shapefile.pdf).
3. Horowitz, E.; Sahni, S. *Fundamentals of data structures*. London: Pitman books Ltd, 1976.
4. MOMIT consortium, 2016. “Multi-scale observation and monitoring of railway infrastructure threats. Home | MOMIT project consortium”. http://www.momit-project.eu/. Accessed: 21 June 2018.
5. The Qt Company, 2018. “Qt | Cross-platform software development for embedded & desktop”. https://www.qt.io/. Accessed: 21 June 2018.
6. Warmerdam, F., 2017. “Shapefile C library”. http://shapelib.maptools.org/. Accessed: 21 June 2018.