UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS DOIS VIZINHOS
CURSO DE ESPECIALIZAÇÃO EM CIÊNCIA DE DADOS

FELIPPE GALDINO SILVA

MANIPULANDO DADOS ESPACIAIS FUZZY EM R USANDO O PACOTE FSR

TRABALHO DE CONCLUSÃO DE CURSO DE ESPECIALIZAÇÃO

DOIS VIZINHOS
2021

FELIPPE GALDINO SILVA

MANIPULANDO DADOS ESPACIAIS FUZZY EM R USANDO O PACOTE FSR

Trabalho de Conclusão de Curso de Especialização apresentado ao Curso de Especialização em Ciência de Dados da Universidade Tecnológica Federal do Paraná, como requisito para a obtenção do título de Especialista em Ciência de Dados.

Orientador: Prof. Dr. Anderson Chaves Carniel

DOIS VIZINHOS
2021

FELIPPE GALDINO SILVA

MANIPULANDO DADOS ESPACIAIS FUZZY EM R USANDO O PACOTE FSR

Trabalho de Conclusão de Curso de Especialização apresentado ao Curso de Especialização em Ciência de Dados da Universidade Tecnológica Federal do Paraná, como requisito para a obtenção do título de Especialista em Ciência de Dados.

Data de aprovação: 17/dezembro/2021

DOIS VIZINHOS
2021

Handling Fuzzy Spatial Data in R Using the *fsr* Package

Anderson Chaves Carniel[*]
Federal University of São Carlos
Department of Computer Science
São Carlos, SP, Brazil
accarniel@ufscar.br

Felippe Galdino
Juliana Strieder Philippsen
Federal University of Technology - Paraná
Dois Vizinhos, PR, Brazil
{ocfgaldino,striederjp}@gmail.com

Markus Schneider
University of Florida
Department of Computer & Information Science & Engineering
Gainesville, FL, USA
mschneid@cise.ufl.edu

ABSTRACT

GIS and spatial data science (SDS) tools have been recently approaching each other by establishing bridge technologies between them. R as one of the most prominent programming languages used in SDS projects has been granted access to GIS infrastructure, while R scripts can be integrated and executed in GIS functions. Unfortunately, the treatment of *spatial fuzziness* has so far not been considered in SDS projects and bridge technologies due to a lack of software packages that can handle *fuzzy spatial objects*. This paper introduces an R package named *fsr* as an implementation of the *fuzzy spatial data types*, *operations*, and *predicates* of the *Spatial Plateau Algebra* that is based on the abstract *Fuzzy Spatial Algebra*. This R package solves the problem of constructing fuzzy spatial objects as *spatial plateau objects* from real datasets and describes how to conduct exploratory spatial data analysis by issuing geometric operations and topological predicates on fuzzy spatial objects. Further, *fsr* provides the possibility of designing *fuzzy spatial inference models* to discover new findings from fuzzy spatial objects. It optimizes the inference process by deploying the *particle swarm optimization* to obtain the point locations with the maximum or minimum inferred values that answer a specific user request.

CCS CONCEPTS

• **Information systems → Geographic information systems**; • **Computing methodologies → Vagueness and fuzzy logic**.

KEYWORDS

Spatial data science, spatial database, spatial fuzziness, Spatial Plateau Algebra, fuzzy spatial inference model, particle swarm optimization

ACM Reference Format:
Anderson Chaves Carniel, Felippe Galdino, Juliana Strieder Philippsen, and Markus Schneider. 2021. Handling Fuzzy Spatial Data in R Using the *fsr* Package. In *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21), November 2–5, 2021, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474717.3484255

[*]This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

1 INTRODUCTION

Several specialized software packages provide sophisticated library support to represent, manipulate, and process *crisp spatial objects* that are characterized by an exact location and a precisely defined extent, shape, and boundary in space. Examples are *GDAL/OGR* (written in C, C++, and Python), *GEOS* (written in C++), *JTS* (written in Java), *Shapely* and *GeoPandas* (written in Python), and *sp*, *sf*, *rgdal*, and *rgeos* (written in R). These software packages are core components in a variety of geographical information systems (GIS), spatial database systems (SDBS), and spatial data science (SDS) projects. They all implement *spatial data types* for the representation of crisp *point*, *line*, and *region* objects and include a large diversity of geometric operations such as topological relationships (e.g., *overlap*, *meet*), geometric set operations (e.g., *intersection*, *union*), and numerical operations (e.g., *length*, *area*).

But increasingly, geoscientists and spatial data scientists are interested in modeling and analyzing spatial phenomena characterized by the feature of *spatial fuzziness*. It captures the inherent property of many spatial objects in reality that have inexact locations, vague boundaries, and/or blurred interiors, and hence cannot be adequately represented by crisp spatial objects. Examples are air polluted areas, soil strata, and habitats of species. In the geoscience and GIS domains, fuzzy set theory has become a popular tool for modeling such *fuzzy spatial objects*. *Fuzzy spatial data types* for *fuzzy points*, *fuzzy lines*, and *fuzzy regions* have been formally defined for representing them. The central idea is to relax the strict decision of belonging (value 1) or non-belonging (value 0) of a point to a spatial object. Instead, partial belonging is allowed and expressed by a *membership degree* in the interval $[0, 1]$. Further, multiple belonging of a point to several spatial objects is allowed with equal or different membership degrees. By analogy to crisp spatial operations, *fuzzy spatial operations* have been defined to process fuzzy spatial objects. Examples are *fuzzy geometric set operations* (e.g., *fuzzy geometric intersection*) and *fuzzy numerical operations* (e.g., *fuzzy area*). By analogy to crisp topological relationships, *fuzzy topological relationships* have been defined to evaluate and term the relative position of two fuzzy spatial objects. Examples are the predicates *fuzzy overlap* and *fuzzy inside*. Such a relationship computes a membership degree between 0 and 1 that indicates to which extent the relationship holds. Such a degree can be mapped to a high-level *linguistic value* and transformed into a Boolean predicate.

The currently most prominent programming languages leveraged in SDS projects are R and Python. Our focus in this paper is on R. Until recently, GIS and SDS tools have been independent of each other although both share common functionalities such as spatial analysis methods. But lately *one-way* and *two-way bridge* technologies have been established between them. GIS tools have begun to provide support for SDS projects in R by granting R access to their infrastructure such as spatial data, spatial algorithms, spatial analysis tools, and spatial visualization concepts (maps). Vice versa, R scripts can be integrated and executed in GIS functions by plugin mechanisms. Examples are (i) the *R-ArcGIS Bridge* to connect R and ArcGIS with the R package *arcgisbinding* in both directions, (ii) the R package *rpostgis* to interact with PostGIS from R, (iii) the procedural language *PL/R* to enable the writing of PostgreSQL functions and triggers in R, (iv) the R packages *RQGIS* and *qgisprocess* to use QGIS as a geoprocessing engine from within R, and (v) a linking mechanism in QGIS to integrate R scripts into QGIS.

Unfortunately, spatial fuzziness has so far not found its way into SDS projects and the aforementioned bridge technologies due to a lack of complete implementations of software packages that can handle fuzzy spatial objects. Motivated by this lack of support, the main contributions of this paper are as follows:

- It introduces an R package named *fsr* as an implementation of the fuzzy spatial data types, operations, and predicates of the *Spatial Plateau Algebra* (*SPA*) that has been designed by the authors and is based on their abstract *Fuzzy Spatial Algebra* (*FUSA*).
- It solves the problem of constructing fuzzy spatial objects as *spatial plateau objects* from real datasets.
- It describes how to conduct exploratory (spatial) data analysis by issuing geometric operations and topological predicates on fuzzy spatial objects.
- It provides the possibility of designing fuzzy spatial inference models to discover new findings from fuzzy spatial objects.
- It optimizes the inference process in the sense that instead of evaluating all points in space, *particle swarm optimization* (*PSO*) is deployed to determine and only evaluate a subset of points that really contributes to finding the final but approximated answer to a user's inference query.

Section 2 discusses related work. Section 3 sketches some basics of fuzzy set theory and fuzzy inference systems. Section 4 introduces a running example to illustrate later how *fsr* works. Section 5 details the concepts and interfaces of the R package *fsr* and focuses on its main classes, its fuzzy spatial operations and predicates, and its fuzzy spatial inference (FSI) model that makes use of PSO. Section 6 draws some conclusions and describes future work.

2 RELATED WORK

Related work can be grouped as follows: (i) R packages for spatial data handling and (ii) implementations of fuzzy spatial objects.

With respect to the first group, we highlight the following R packages[1]: *rgeos*, *rgdal*, *sp* [13], and *sf* [12]. Their common characteristic is to provide interfaces, bindings, and bridges to spatial libraries written in C/C++. Hence, it allows the use of functionalities provided by well-known spatial libraries in R scripts and applications. While *rgeos* and *rgdal* are interfaces to GEOS[2] and GDAL[3], respectively, *sp* and *sf* define classes and methods for handling and visualizing spatial data in R. The latter two packages also use underlying libraries to process specific spatial operations (e.g., topological relationships). *sf* distinguishes itself due to its (i) broad collection of versatile functions, (ii) compatibility with OGC standards and SDBS like PostgreSQL/PostGIS[4], and (iii) compatibility with R packages for data science [19] (i.e., *tidyverse*). Unfortunately, these R packages do not offer support for fuzzy spatial data since they focus on dealing with crisp spatial objects only.

Table 1: Comparison of *fsr* with related work.

Comparison criteria	[9]	[17]	[1]	[7, 8]	*fsr*
Fuzzy points	✓		✓		✓
Fuzzy lines	✓		✓		✓
Fuzzy regions	✓	✓		✓	✓
Visualization methods	✓		✓*		✓
Geometric set operations	✓*		✓		✓
Numerical operations					✓
Topological relationships				✓	✓
Fuzzy spatial inference					✓
Programming language	unknown	unknown	C	PL/pgSQL	R

* limited support or only partial implementation of the operations

The second group comprises implementations of fuzzy spatial data types and fuzzy spatial operations. Table 1 compares them with *fsr*. Each implementation deploys a specific conceptual model that formally defines what fuzzy spatial objects are and how they can be manipulated by leveraging point set theory, fuzzy set theory, and fuzzy topology. Dilo et al. [9] take the conceptual model in [10] to implement simple fuzzy points, simple fuzzy lines, and simple fuzzy regions as an extension of the GRASS GIS. The authors provide visualization methods in the form of a GRASS module and implement geometric set operations (except the difference between fuzzy lines). Verstraete [17] proposes implementable representation methods for fuzzy regions, such as bitmap representations [18] and TINs. Carniel et al. [1] introduce a PostgreSQL/PostGIS extension to deal with fuzzy points and fuzzy lines according to the model in [10]. Davari and Ghadiri [7, 8] specify fuzzy regions and fuzzy spatial skyline queries in PL/pgSQL by using existing functionalities of PostgreSQL/PostGIS. Further, fuzzy topological relationships use the concepts of the Fuzzy Region Connection Calculus [16].

The aforementioned approaches face at least one of the following problems: (i) low applicability due to the limited support for fuzzy spatial data types and operations, and (ii) problems inherited from their conceptual models, such as a limited representation of spatial fuzziness and lack of closure properties (e.g., as discussed in [2]). On the other hand, *fsr* does not have these drawbacks because it implements *all* concepts defined by SPA [4, 15] and FUSA [2], which specify data structures for fuzzy spatial data types and define a broad collection of fuzzy spatial operations. *fsr* goes further since it provides functions for designing fuzzy spatial inference models on fuzzy spatial objects. This is an additional functionality that expands the range of applications (e.g., SDBS, GIS, SDS) in which *fsr* can be used.

[1]The webpages for these R packages have the canonical form https://CRAN.R-project.org/package=*name*, where *name* should be replaced with the package name.
[2]https://trac.osgeo.org/geos
[3]https://gdal.org/
[4]https://postgis.net/

3 FUZZY SETS AND LINGUISTIC VARIABLES

Fuzzy set theory [20] extends and generalizes Boolean set theory. Let X be the *universe* (*of discourse*). Let A be a classical subset of X. The membership of an element to A is given by the *characteristic function* $\chi_A : X \longrightarrow \{0, 1\}$ such that for all $x \in X$ holds that $\chi_A(x) = 1$ if, and only if, $x \in A$, and $\chi_A(x) = 0$ otherwise. Fuzzy set theory relaxes this strict rule by allowing that an element can have partial membership in the set. Let $\tilde{A}$ be a fuzzy set. All elements in X receive an evaluation regarding their membership in $\tilde{A}$ by using the *membership function* $\mu_{\tilde{A}} : X \longrightarrow [0, 1]$. Larger (smaller) values mean higher (lower) degrees of set membership. Hence, fuzzy set theory permits that an element has multiple and different *membership degrees* in different fuzzy sets.

Applications represent the vagueness of a concept by using *linguistic values* (LVal) in the scope of a *linguistic variable* (LVar) [21]. While LVars are attributes, LVals characterize situations of an LVar. For instance, for a given LVar named *temperature level*, we can distinguish situations where a temperature value is characterized by an LVal, such as *cold*, *warm*, or *hot*. LVals are labels for fuzzy sets in applications. In this paper, we discuss that fuzzy spatial objects implemented as spatial plateau objects can represent LVals.

4 RUNNING EXAMPLE

Throughout the rest of this paper, we build up a small application to illustrate how *fsr* works. For this, we make use of the following real spatial datasets: (i) Airbnb accommodations in New York City, extracted from 2021-04-07 to 2021-04-12 and named *accom_nyc_full*[5], and (ii) New York City restaurant inspection results provided by the Department of Health and Mental Hygiene (DOHMH) named *rest_nyc_full*[6]. Each dataset contains crisp point objects labeled with several alphanumerical attributes. We have performed filters on these datasets to produce the datasets of our running example named *accom_nyc* and *rest_nyc* as follows. *accom_nyc* contains Airbnb accommodations that have overall ratings; thus, it is a subset of *accom_nyc_full* and excludes observations with missing data in this attribute. *rest_nyc* comprises the most recent graded inspection results valid on April 12th, 2021, since a restaurant can be reinspected. It is a subset of *rest_nyc_full* and results from the execution of an R script provided DOHMH[6] and the deletion of observations with negative scores and missing coordinates. In the end, *accom_nyc* and *rest_nyc* have 26,496 and 24,699 lines, respectively.

We use some attributes from them only. Each attribute is represented as an LVar with a set of LVals that characterize the values of the attributes. *accom_nyc* contains two numerical attributes of interest for this application. The first one represents the daily price of the accommodation and its LVar is named *accommodation price*, which can be characterized as *cut-rate*, *affordable*, and *expensive*. The second one stores the overall rating as a value from 0 (worst) to 100 (best). We name its LVar *accommodation review* and describe its LVals *reasonable*, *good*, and *excellent*. As for *rest_nyc*, we are interested in the sanitary inspection results represented as scores greater than or equal to 0, where a smaller score implies a better grade. Based on that, we specify the LVar *food safety* with the following levels of sanitary conditions: *low*, *medium*, and *high*.

The goal of our application is fourfold. First, we aim to build spatial plateau objects for representing the desired LVals of each dataset. For instance, we wish to create a plateau region object that represents the areas containing *cut-rate accommodations*. Second, we conduct an exploratory spatial data analysis by performing spatial plateau operations on the built plateau region objects. Third, we design a fuzzy spatial inference model to estimate the *visiting experience* based on prices and overall ratings of accommodations as well as sanitary conditions of restaurants. The output of such a model infers a value between 0 and 100 that indicates how attractive it is to visit a specific location. For this, we classify the experience as *awful*, *average*, and *great*. Finally, we aim to obtain only those locations that promise to provide a *great visiting experience*.

[5]http://insideairbnb.com/get-the-data.html
[6]https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/43nn-pn8j

5 *FSR* PACKAGE

The *fsr* package is an R implementation of the Spatial Plateau Algebra (SPA) that is based on the abstract and conceptual Fuzzy Spatial Algebra (FUSA). It is publicly available at https://cran.r-project.org/package=fsr. Its *spatial plateau data types*, *spatial plateau operations*, and *spatial plateau predicates* are defined in terms of the spatial data types, operations, and predicates of crisp spatial algebras (Section 2) for which implementations are available. Therefore, the SPA can be regarded as an *executable type system*.

Section 5.1 provides an architectural overview of *fsr*. Section 5.2 informally describes the spatial plateau data types, introduces a format for their textual representation, and shows their use in *fsr*. Section 5.3 deals with a systematic two-stage approach to creating spatial plateau objects from real spatial datasets and demonstrates how this approach is applied in *fsr* in the context of our running example. Section 5.4 sketches the spatial plateau operations and predicates of the SPA and illustrates their application in *fsr*. Section 5.5 describes the design and evaluation of fuzzy spatial inference models by using FIFUS [3].

5.1 Architectural Overview

Figure 1 presents the overview of the *fsr* package. Applications like SDBS, GIS, and SDS, make use of the *fsr* package by calling functions available in the four modules named *Basic Module* (Section 5.2), *Construction Module* (Section 5.3), *Fuzzy Spatial Data Handling Module* (Section 5.4), and *Fuzzy Spatial Inference Module* (Section 5.5).

We employ other R packages to implement *fsr*. Some of them are shown in the bottom layer of Figure 1. The *R Base* package provides the basic programming support for R. The *methods* package allows us to define R classes and their formal methods. Since SPA uses a crisp spatial type system to specify fuzzy spatial objects as plateau spatial objects, we deploy the *sfg* data type from the *sf* package to implement and handle crisp spatial objects. We use the *FuzzyR* [6] package to deal with membership functions when creating spatial plateau objects and designing fuzzy spatial inference models. The *tidyverse* package contains a set of other packages which we use for handling strings and tabular data and for producing graphical visualizations of spatial plateau objects. Finally, the *pso* package supplies the implementation of the PSO algorithm. Other packages are also employed but not listed in the figure.

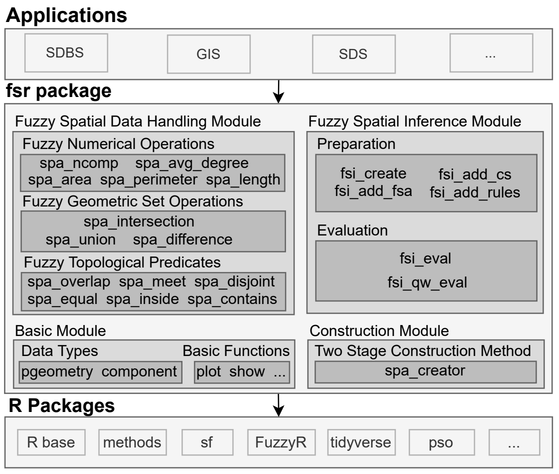


Figure 1: An overview of the *fsr* package, its underlying R resources, and some possible applications.

5.2 Classes for Spatial Plateau Data Types

Concepts. The SPA represents fuzzy points, fuzzy lines, and fuzzy regions as (*spatial*) *plateau points*, *plateau lines*, and *plateau regions*, respectively. In general, a spatial plateau object is specified by a list of pairs where each pair (called *component*) consists of a crisp spatial object and a membership degree in $]0, 1]$. The crisp spatial objects of all components of a spatial plateau object must be of the same crisp spatial data type *point*, *line*, or *region*, have different membership degrees, and be disjoint or adjacent to each other. Examples of spatial plateau objects are given in Figure 2. A formal definition of the spatial plateau data types is given in [4, 15].

The membership degree of a single point in a spatial plateau object depends on the spatial plateau data type. Each component of a plateau point object consists of a single point and an associated membership value. The associated membership degree of each region component of a plateau region is assigned to all interior points of that component. However, each boundary point of a region component gets the maximum membership degree of all region components to which it belongs. Similarly, each point of a component of a plateau line object obtains the maximum membership degree of all line components to which it belongs.

The textual representation of a spatial plateau object has been introduced in [5] and combines the well-known text (WKT) representation for crisp vector geometry objects and the formal definitions of the three spatial plateau data types. This textual representation called *PWKT* is specified by a function *pwkt* that takes a spatial plateau object as input and generates a string as output. Let us assume a spatial plateau region object *pr* whose *n* region components are named r_i and have the membership degree m_i for $i \in \{1, \ldots, n\}$. If *wkt* is the function that generates the WKT representation of a complex region object, *pwkt* applied to *pr* generates

$$pwkt(pr) = \texttt{"PLATEAUREGION(("} + wkt(r_1) + \texttt{", "} + m_1 + \texttt{"), "} \\ + \ldots + \texttt{", ("} + wkt(r_n) + \texttt{", "} + m_n + \texttt{"))"}$$

where the "+" symbol denotes string concatenation and the membership value is implicitly transformed into a string.

Running Example Using *fsr*. We employ the S4 object-oriented system of R to implement the underlying classes of *fsr*. This system allows us to explicitly define classes with a particular number of attributes so that we can create objects through the constructor function *new*. We define two S4 classes. The first one is named *component* and has two attributes that represent a pair composed of a crisp spatial object and a membership degree in $]0, 1]$. Since we are using the *sf* package, the crisp spatial object is represented as an *sfg* object. This means a component stores a single instance of a simple or complex crisp spatial object.

The second class of *fsr* is called *pgeometry*, which stands for *plateau geometry object*. Its attributes include a list of *component* objects, an *sfg* object that represents the *support* (i.e., the union of the crisp spatial objects of all components) of the *pgeometry* object, and its data type. In particular, the storage of the support improves the computation of spatial plateau operations (Section 5.4).

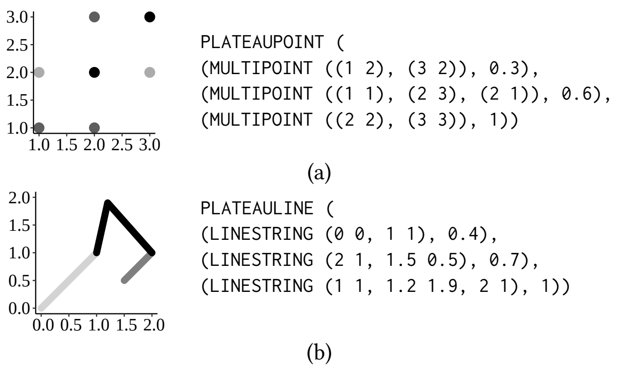
The *fsr* package provides several constructors that enable us to create *component* and *pgeometry* objects. Here, we present two constructors named *component_from_sfg* and *create_pgeometry* with the following signatures:

```
component_from_sfg(sfg, md)
create_pgeometry(components, type)
```

The constructor *component_from_sfg* receives an *sfg* object and a membership degree as inputs and yields a *component* object. The constructor *create_pgeometry* accepts a list of components and a string value that indicates the desired spatial plateau data type as inputs and yields a *pgeometry* object. These constructors ensure that the properties defined by SPA are fulfilled. For instance, *component_from_sfg* creates a *component* object only if a simple or complex point, line, or region object is provided. Further, it is unable to create a *pgeometry* object with a list of components whose *sfg* objects are not of the same simple or complex spatial data type.

We also set methods that override well-known functions of R to deal with *pgeometry* objects. We highlight two methods. The method *show* is responsible for providing textual representations of objects so that programmers can view the value of an object in the console. In this case, *fsr* produces the PWKT representation of a *pgeometry* object. The second method *plot* provides graphical visualizations of objects. We supply this method in *fsr* by using the *ggplot2* package to visualize the components of a *pgeometry* object. The default behavior is to use grayscale to indicate how much a point belongs to the *pgeometry* object where darker colors mean higher membership degrees. We also provide the function *fsr_plot* that allows users to specify other visualization options.

Program 1 shows a small example that uses the aforementioned functions. It starts by defining a set of matrices containing the coordinates to build crisp point objects and crisp line objects (lines 1 to 6). Next, we create three components for these crisp objects (lines 7 to 12). To build the crisp spatial objects, we employ the functions *st_multipoint* and *st_linestring* from *sf*. Then, the program builds a plateau point object and a plateau line object (lines 13 and 14). The surrounding parentheses in the same lines implicitly call the *show* method to print them using their PWKT representations. Finally, such objects are graphically visualized by using the function *plot* (lines 15 and 16). The graphical and textual representations of these objects are shown in Figure 2.



```
PLATEAUPOINT (
(MULTIPOINT ((1 2), (3 2)), 0.3),
(MULTIPOINT ((1 1), (2 3), (2 1)), 0.6),
(MULTIPOINT ((2 2), (3 3)), 1))
```

```
PLATEAULINE (
(LINESTRING (0 0, 1 1), 0.4),
(LINESTRING (2 1, 1.5 0.5), 0.7),
(LINESTRING (1 1, 1.2 1.9, 2 1), 1))
```

Figure 2: Examples of a plateau point object (a) and a plateau line object (b). Their creation is shown in Program 1.

Program 1: Using basic functions of *fsr*.

```
pts1 <- rbind(c(1, 2), c(3, 2))
pts2 <- rbind(c(1, 1), c(2, 3), c(2, 1))
pts3 <- rbind(c(2, 2), c(3, 3))
lpts1 <- rbind(c(0, 0), c(1, 1))
lpts2 <- rbind(c(1, 1), c(1.2, 1.9), c(2, 1))
lpts3 <- rbind(c(2, 1), c(1.5, 0.5))
cp1 <- component_from_sfg(st_multipoint(pts1), 0.3)
cp2 <- component_from_sfg(st_multipoint(pts2), 0.6)
cp3 <- component_from_sfg(st_multipoint(pts3), 1)
cp4 <- component_from_sfg(st_linestring(lpts1), 0.4)
cp5 <- component_from_sfg(st_linestring(lpts2), 1)
cp6 <- component_from_sfg(st_linestring(lpts3), 0.7)
(pp <- create_pgeometry(list(cp1,cp2,cp3), "PLATEAUPOINT"))
(pl <- create_pgeometry(list(cp4,cp5,cp6), "PLATEAULINE"))
plot(pp)
plot(pl)
```

5.3 Creating Spatial Plateau Objects from Real Spatial Datasets

Concepts. The construction of spatial plateau objects from real datasets can only be successful with a systematic approach. Our approach rests on a two-stage construction method. The input is a point dataset where each point represents the location of a phenomenon treated by the application. Each point is annotated with alphanumerical data that describe its meaning in the application. This dataset is used by the first stage called *fuzzification stage*. This stage assigns membership degrees to each point of the dataset according to the requirements of the application. Each membership degree indicates how much a point belongs to a particular LVal (e.g., cut-rate, affordable, expensive) as a possible characterization of an LVar (e.g., accommodation price). Since different ways of assigning membership degrees to points are conceivable, this stage includes a *fuzzification policy*. The main advantage of this strategy is that users can choose the best policy that fulfills their needs. Three general fuzzification policies are offered that are based on fuzzy sets, fuzzy clustering algorithms, and fuzzy classification algorithms.

In our running example, we employ the fuzzy set policy, which requires the specification of membership functions to represent an LVal. Examples of membership functions include the *triangular membership function* (TrimMF) and the *trapezoidal membership function* (TrapMF) [20].

The second stage called *construction stage* is responsible for grouping the points belonging to the same LVal into a corresponding plateau region object. The construction stage includes a *construction policy* that lets the user decide how the plateau region objects are spatially defined. Construction policies are either based on Voronoi diagrams, Delaunay triangulations, or convex hulls. The last step of this stage returns tabular data in which each tuple represents an LVal and its correspondingly created plateau region object. This strategy permits that users manage and query these objects using the SPA (Section 5.4). More details about this two-stage construction method can be found in [5].

Program 2: Constructing plateau region objects for the LVar *accommodation price.*

```
lvals_accom_price <- c("cut-rate", "affordable", "expensive")
cut_rate_mf <- genmf("trapmf", c(0, 0, 10, 48))
affordable_mf <- genmf("trapmf", c(10, 48, 80, 115))
expensive_mf <- genmf("trapmf", c(80, 115, 10000, 10000))
accom_price <- accom_nyc[, c("longitude", "latitude", "price")]
accom_price_layer <- spa_creator(accom_price, classes =
   lvals_accom_price, mfs = c(cut_rate_mf, affordable_mf,
   expensive_mf))
# plotting cut-rate, affordable, and expensive accommodations
plot(accom_price_layer$pgeometry[[1]], color=NA, base_poly=nyc)
plot(accom_price_layer$pgeometry[[2]], color=NA, base_poly=nyc)
plot(accom_price_layer$pgeometry[[3]], color=NA, base_poly=nyc)
```

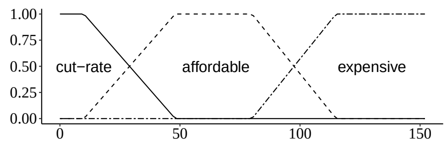


Figure 3: TrapMFs for the LVar *accommodation price.*

Running Example Using *fsr*. Since the LVars *accommodation price*, *accommodation review*, and *food safety* contain geographical coordinates, we can create spatial region objects that characterize their LVals by using our two-stage construction method. The package *fsr* implements this method through the function *spa_creator*, which has the following signature:

```
spa_creator(tbl, fuzz_policy = "fsp",
            const_policy = "voronoi", ...)
```

The first parameter *tbl* denotes the point dataset annotated with numerical values. The second and third parameters *fuzz_policy* and *const_policy* allow users to choose the fuzzification and construction policies. The default policies are the fuzzy set policy (*fsp*) and the Voronoi diagram policy (*voronoi*). If a policy requires additional parameters, they have to be listed in the variable parameter list indicated by the "three dots" argument. For instance, the fuzzy set policy requires a character vector containing the names of LVals and another vector containing its corresponding membership functions. The output of *spa_creator* is a table containing the plateau region objects labeled with their LVals.

Program 2 applies the function *spa_creator* for our running example. It has two parameters. First, it requires a character (string)

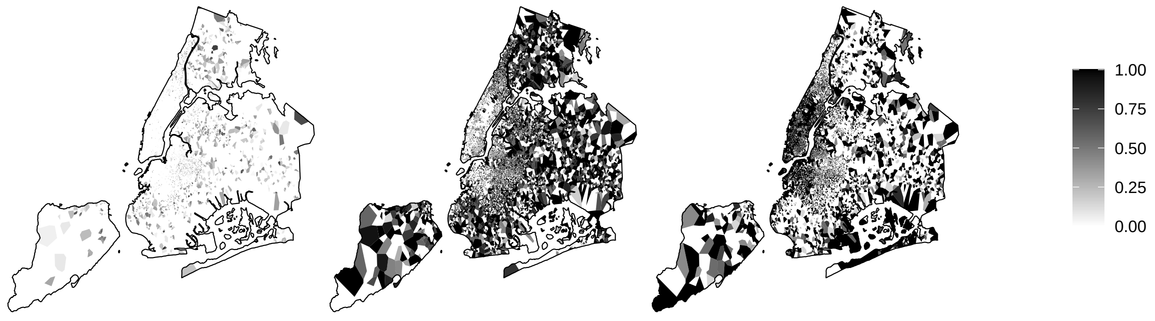


Figure 4: The plateau region objects for each LVal *cut-rate* (a), *affordable* (b), and *expensive* (c) of the LVar *accommodation price* including city boundaries.

vector containing all LVals of *accommodation price* (line 1). Second, it expects the membership functions that characterize each LVal (lines 2 to 4). For this, the function *genmf* from the FuzzyR package generates four TrapMFs that are shown in Figure 3. Then, we select only the columns from *accom_nyc* that refer to the coordinates of Airbnb accommodations and their daily prices (line 5). This selection is stored in *accom_price*. Next, we call *spa_creator* to build plateau regions by taking *accom_price* as first input. Since Program 2 employs the default policies, we include the character vector with the LVal names and a vector of the generated membership functions as inputs (line 6). Both vectors are correlated in the sense that the element *i* of one vector corresponds to the element *i* of the other vector. Finally, the program provides visualizations of the created plateau region objects (lines 8 to 10). For this, the function *plot* is used with two parameters that remove the colors of boundaries of the components of a *pgeometry* object (`color = NA`) and add the New York City boundaries (`base_poly = nyc`, where `nyc` is an *sfg* object) to the visualization, as shown in Figure 4. In Figure 4b, for any specific point location with membership degree 1, the accommodation price is between 48 and 80 since the TrapMF for the linguistic value *affordable* returns 1 for these values (Figure 3).

Similarly to Program 2, we create the plateau region objects for the other LVars of our running example. This leads to the generation of the three layers *accom_price_layer*, *accom_review_layer*, and *food_safety_layer*, respectively, corresponding to the LVars *accommodation price*, *accommodation review*, and *food safety*. To build the layer *accom_review_layer*, we specify the TrapMFs (0, 0, 40, 65), (40, 65, 80, 85), and (80, 85, 100, 100) for the LVals *reasonable*, *good*, and *excellent*, respectively. To create the layer *food_safety_layer*, we use the TrapMFs (24, 28, 115, 115), (10, 14, 24, 28), and (0, 0, 10, 14) for the LVals *low*, *medium*, and *high*, respectively. As a result, each layer consists of a set of plateau region objects labeled with LVals. Figure 5a depicts the plateau region object for high food safety.

5.4 Manipulating Spatial Plateau Objects

Concepts. SPA [4, 15] contains three classes of spatial plateau operations. The first class of *spatial plateau set operations* comprises the functions *spa_intersection*, *spa_union*, and *spa_difference* that implement the respective geometric operations *fintersection*, *funion*, and *fdifference* of FUSA. They all have the signature $\alpha \times \alpha \times \beta \rightarrow$

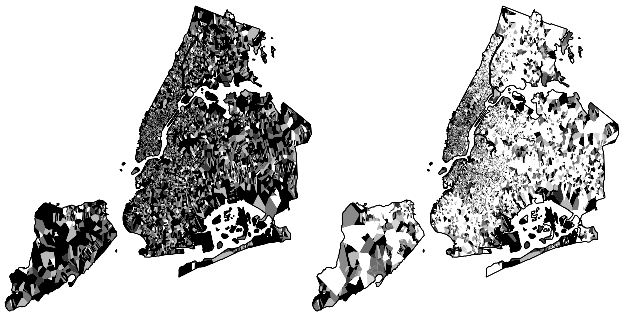


Figure 5: The plateau region object for the LVal *high* (a) of the LVar *food safety* and its intersection with the plateau region object of Figure 4c (b). This means that each point in (b) represents to which extent the accommodation is expensive *and* the food safety is high in the particular location.

α with $\alpha \in \{\texttt{PLATEAUPOINT}, \texttt{PLATEAULINE}, \texttt{PLATEAUREGION}\}$ and $\beta \in \{itype, utype, dtype\}$. This means that the spatial plateau objects that form the input and the output of each operation are of the same spatial plateau data type. The types *itype* for the operation *spa_intersection*, *utype* for the operation *spa_union*, and *dtype* for the operation *spa_difference* contain the names of possible operators that can be the third parameter of the respective operation. Their selection depends on the application and is therefore *context-dependent*. Type *itype* contains the names of *t-norms* (e.g., *min* and *prod*), type *utype* the names of *t-conorms* (e.g., *max*), and type *dtype* the names of *difference operators* (e.g., *absolute difference*, *bounded difference*). All these operators have the signature $[0, 1] \times [0, 1] \rightarrow [0, 1]$ and calculate the membership degree of a point of the resulting spatial plateau object from the membership degrees of the same point in the two input spatial plateau objects.

The second class of *spatial plateau metric operations* offers the type-independent functions *spa_ncomp* and *spa_avg_degree* to determine the number of components and calculate the average membership degree of a spatial plateau object respectively. The type-dependent functions *spa_perimeter* and *spa_area* compute the area

Program 3: Handling spatial plateau objects by using spatial plateau operations.

```
1 intersected_area <-
    spa_intersection(accom_price_layer$pgeometry[[3]],
    food_safety_layer$pgeometry[[3]])
2 plot(intersected_area, color = NA, base_poly = nyc)
3 total_area <- spa_area(intersected_area)
4 overlap_degree <- spa_overlap(accom_review_layer$pgeometry[[3]],
    food_safety_layer$pgeometry[[1]]) # excellent & low
5 is_quite_overlap <- spa_overlap(accom_price_layer$pgeometry[[2]],
    accom_review_layer$pgeometry[[3]], ret = "bool", lval = "quite",
    eval_mode = "soft_eval") # affordable & excellent
```

and the perimeter of a plateau region object, respectively, and *spa_length* computes the length of a plateau line object.

The third class relates to *spatial plateau topological relationships*. Any two spatial plateau objects are in a particular relative position to each other that can be expressed by a *topological relationship* such as *overlap* or *inside*. Such a topological relationship determines the degree to which it holds for any two spatial plateau objects by a real value in the interval $[0, 1]$. The topological relationships provided are *spa_overlap*, *spa_meet*, *spa_disjoint*, *spa_equal*, *spa_inside*, and *spa_contains* on two plateau regions. Their key idea is to consider point subsets resulting from the combination of spatial plateau set operations and spatial plateau metric operations on the spatial plateau objects for computing the resulting degree. Their definitions are complex and can be found in [4].

Running Example Using *fsr*. The *fsr* package provides R implementations of all three classes of spatial plateau operations. The spatial plateau set operations have the signatures:

```
spa_intersection(pgo1, pgo2, itype = "min")
spa_union(pgo1, pgo2, utype = "max")
spa_difference(pgo1, pgo2, dtype = "f_diff")
```

The first two parameters of these functions are *pgeometry* objects of the same type. The third parameter allows the user to specify an operator name from the types *itype*, *utype*, or *dtype* respectively that will be used to evaluate the operation. The *fsr* package provides some operators by default and permits the specification and implementation of user-defined operators. The latter must have the correct signature and fulfill the required properties of the respective operator type. To employ a user-defined operator, its name must be provided as the third parameter of a spatial plateau set operation.

The *fsr* package provides the spatial plateau metric operations with the signatures:

```
spa_ncomp(pgo)
spa_avg_degree(pgo)
spa_perimeter(pr)
spa_area(pr)
spa_length(pl)
```

Each operation validates if the *pgeometry* object given as its input has the correct data type. If not, it throws a warning message and returns 0, similarly to the type-dependent operations implemented by the *sf* package. The *fsr* package provides the following spatial plateau topological relationships in R with the signatures

```
spa_overlap(pgo1, pgo2, itype = "min", ret = "degree", ...)
spa_meet(pgo1, pgo2, itype = "min", ret = "degree", ...)
spa_disjoint(pgo1, pgo2, itype = "min", ret = "degree", ...)
spa_equal(pgo1, pgo2, utype = "max", ret = "degree", ...)
spa_inside(pgo1, pgo2, utype = "max", ret = "degree", ...)
spa_contains(pgo1, pgo2, utype = "max", ret = "degree", ...)
```

The first two parameters of these functions are *pgeometry* objects of the type `PLATEAUREGION`. Since the spatial plateau topological relationships deploy geometric set operations in their implementations, the third parameter permits that users apply a specific operator to the internal set operation of the topological relationship. A t-norm can be provided to the functions *spa_overlap*, *spa_meet*, and *spa_disjoint*, while a t-conorm can be given for the other functions.

The fourth parameter defines the returning value of a function, and its possible values are: *degree* (default), *list*, and *bool*. If the parameter value is *degree*, the function returns a value of the interval $[0, 1]$ indicating the degree of truth of a given topological relationship. For the remainder options, the functions utilize a set of LVals that characterize the different situations of topological relationships. For instance, we can distinguish if two plateau region objects are *somewhat* or *slightly* overlapping. Each LVal has its corresponding membership function defined in the domain $[0, 1]$.

If the parameter value is *list*, the function returns a named list containing how much the result of the predicate belongs to each LVal. Since fuzzy set theory allows that an element has different membership degrees in different sets, this parameter allows users to analyze the spatial vagueness expressed by a topological predicate. For instance, the overlapping situation of two plateau region objects can be 0.4 for *somewhat* and 0.6 for *slightly*.

Finally, if the parameter value is *bool*, the function returns a Boolean value indicating whether the degree returned by the topological relationship matches a given LVal according to an *evaluation mode*. In this case, the names of the LVal and of the evaluation mode have to be specified by the "three dots" argument. The *fsr* package provides several evaluation modes. An example is the evaluation mode *soft_eval*, which returns *true* if the degree returned by the topological relationship has a membership greater than 0 in the membership function of a given LVal; otherwise, it returns *false*.

User-defined LVals and membership functions for topological relationships are specified by the following function:

```
spa_set_classification(classes, mfs)
```

The parameter *classes* is a character vector containing the LVals, and the parameter *mfs* is the corresponding vector of membership functions generated by the function *genmf* of the FuzzyR package. In addition, user-defined evaluation modes can be implemented as functions with the signature $[0, 1] \rightarrow \{true, false\}$.

Program 3 depicts a short example that manipulates the spatial plateau objects of our running example. First, we identify the areas containing expensive accommodations and restaurants with high food safety. For this, we compute the intersection of the plateau region objects of the respective LVals (line 1). Then, we visualize the resulting spatial plateau object (line 2) as shown in Figure 5b. Next, we measure the area of this object (line 3). In line 4, we check the overlapping degree between excellent accommodations and restaurants with low food safety. Finally, we evaluate whether affordable accommodations are quite overlapping with excellent accommodations (line 5). For this, we specify the needed parameters to use the topological relationship as a Boolean predicate (i.e., `ret = "bool"`, `lval = "quite"`, and `eval_mode = "soft_eval"`).

5.5 Processing Fuzzy Spatial Inference Models

Concepts. LVars and LVals enable us to write fuzzy conditional propositions called *fuzzy rules* that express expert knowledge of the application domain [21]. A fuzzy rule has the format *IF A THEN B*, where *A* is called the *antecedent* and *B* the *consequent* of the rule such that *A* implies *B*. *A* and *B* are statements of the format *L is v* where *L* is an LVar and *v* is an LVal of the scope of *L*. Statements can be combined by using logical connectives like AND and OR.

FIFUS [3] enables us to create fuzzy spatial inference (FSI) models based on traditional fuzzy inference systems [11]. It gets a crisp simple point object as input and yields the reasoning conclusion for that point, such as a recommendation, estimation, or prediction. For this, it requires two components. The *data source component* contains the needed elements for performing an FSI: (i) fuzzy spatial objects labeled with LVals, (ii) fuzzy sets representing the expected results of the application, and (iii) the fuzzy rules set. In FIFUS, each statement of the antecedent of a fuzzy rule employs the LVals of the fuzzy spatial objects since they represent the main characteristics of the application; thus, each fuzzy rule has a *fuzzy spatial antecedent* (FSA). Each statement of the consequent makes use of the membership function that represents the employed LVal.

The *spatial reasoning component* is responsible for evaluating a fuzzy inference method (e.g., Mamdani's method [11]) on the point object given as input by using the elements of the data source component. In general, an FSI model performs the following steps. For each fuzzy rule, it determines the degree to which the point object belongs to each of the fuzzy spatial objects representing the LVals of the FSA. If the FSA has more than one statement, then it combines the degrees by using operators corresponding to the employed logical connectives. The result of this procedure is the degree of fulfillment of the rule (also known as *firing strength* of the rule). Then the inference method applies an *implication* operator that reshapes the fuzzy set of the consequent. The resulting implications are aggregated by a *composition* operator. The underlying inference method can apply different operators to the logical connectives, one operator for the implication, and one operator for the composition. For instance, Mamdani's method applies the *min* operator for the AND operator and the implication, and the *max* operator for the OR operator and the composition. The result of the composition is a fuzzy set without meaning for the user. Hence, the last step of the FSI is to extract a numeric value that best represents the resulting fuzzy set by executing a *defuzzification* method such as the *centroid*. More details about FIFUS are given in [3].

However, the usability of an FSI model is limited if we consider a crisp simple point object as input only. Instead, we consider a *query window* as input. A query window is an axis-aligned rectangle that represents an infinite set of points in the Euclidean plane. For instance, in the context of our running example, the user wants to know the locations where the visiting experience is great in a given query window of interest. This leads to a kind of inference that we call *query window inference*. Since it is infeasible to infer all points in the query window due to their infinite number, we propose two approaches that answer different types of questions.

The first approach is to determine a sample by extracting a finite number of points from the query window to be evaluated by the FSI. We call this approach the *discretization approach*. It is an approach that requires the number k of points to be extracted as a "hyperparameter" whose value is used to control the query window inference. This approach can answer questions like "return all the $m \leq k$ points with inferred values belonging to a target LVal". In our running example, we are interested in capturing all the point locations with *great* visiting experience. This approach can also answer the question "return the point locations with the maximum (minimum) inferred value". For this, we apply a filter to get the $n \leq m$ points with the maximum (minimum) inferred value. However, several points in the query window that are not among the k points can have the maximum (minimum) inferred value so that this approach cannot discover them.

The second approach is a better alternative to answer the last question. It avoids the drawback of the discretization approach by identifying and processing a subset of points of the query window that definitely contributes to answering the question. In our running example, users are interested in gathering only the locations whose visiting experience has the highest rating so that other locations are not relevant for them. For this purpose, we can employ an optimization technique to obtain an approximate solution that maximizes the output value of an FSI model.

In this paper, we employ PSO [14] to implement the second approach since it requires a small number of parameters and has a relatively small computational cost compared to other optimization techniques (e.g., genetic algorithms). PSO is based on swarm intelligence and aims to optimize a given problem iteratively by considering a search space and the individual and collective performance of *particles* in this space. A particle can be defined as a possible solution to the problem to be optimized. Each particle evaluates a *fitness function* at its current position [14]. Its movement in the search space is then determined by using solutions found by itself (best local) and by other particles (best global) [14]. In addition to the fitness function, PSO requires the definition of an initial quantity of particles (i.e., a population) that will travel through the search space to find the best solution for the problem. Further, the maximum number of iterations is a common parameter.

Running Example Using *fsr*. The usage of FSI models is subdivided into a *preparation phase* and an *evaluation phase*. The preparation phase is responsible for instantiating a new FSI model with the elements of the data source component of FIFUS. For this, the *fsr* package provides the following functions:

```
fsi_create(name, and_method = "min", or_method = "max",
        imp_method = "min", agg_method = "max",
        defuzz_method = "centroid", default_conseq = NULL)
fsi_add_fsa(fsi, lvar, tbl)
fsi_add_cs(fsi, lvar, lvals, mfs, bounds)
fsi_add_rules(fsi, rules, weights = rep(1, length(rules)))
```

The first function creates an FSI model without elements of the data source component. Its first parameter specifies the name of the FSI model. The next four parameters define the operators for the logical connectives AND and OR, the implication operator, and the aggregation operator, respectively. The default values for these parameters lead to the use of Mamdani's method for the FSI model. The parameter *defuzz_method* determines the defuzzification technique and its default value is the *centroid* technique. Other defuzzification techniques in the FuzzyR package can also be specified. The last parameter is a membership function generated by

Program 4: FSI model for our running example.

```
1  fsi <- fsi_create("To visit or not to visit, that is the question",
     default_conseq = genmf("trimf", c(10, 30, 60)))
2  fsi <- fsi_add_fsa(fsi, "accommodation price", accom_price_layer)
3  fsi <- fsi_add_fsa(fsi,"accommodation review",accom_review_layer)
4  fsi <- fsi_add_fsa(fsi, "food safety", food_safety_layer)
5  lvals_visiting_exp <- c("awful", "average", "great")
6  awful_mf <- genmf("trimf", c(0, 0, 20))
7  average_mf <- genmf("trimf", c(10, 30, 60))
8  great_mf <- genmf("trapmf", c(40, 80, 100, 100))
9  fsi <- fsi_add_cs(fsi, "visiting experience", lvals_visiting_exp,
     c(awful_mf, average_mf, great_mf), c(0, 100))
10 rules <- c("IF accommodation review is reasonable AND food safety
     is low THEN visiting experience is awful",
     "IF accommodation price is expensive AND accommodation review is
     reasonable THEN visiting experience is awful",
     "IF accommodation price is affordable AND accommodation review is
     good AND food safety is medium THEN visiting experience is
     average",
     "IF accommodation price is affordable AND accommodation review is
     excellent AND food safety is high THEN visiting experience is
     great",
     "IF accommodation price is cut-rate AND accommodation review is
     excellent AND food safety is high THEN visiting experience is
     great")
11 fsi <- fsi_add_rules(fsi, rules)
12 res <- fsi_eval(fsi, st_point(c(-73.992, 40.7145)))
13 pts_qw1 <- rbind(c(-73.92, 40.68527), c(-73.75, 40.68527),
     c(-73.75, 40.75), c(-73.92, 40.75), c(-73.92, 40.68527))
14 qw1 <- st_polygon(list(pts_qw1))
15 dis_res <- fsi_qw_eval(fsi, qw1, approach = "discretization",
     target_lval = "great", k = 100)
16 pso_res <- fsi_qw_eval(fsi, qw1, approach = "pso", max_depth = 2)
```

the function *genmf* of the FuzzyR package and defines the default behavior of the FSI model. That is, if there is no fuzzy rule with a degree of fulfillment greater than 0, the membership function *default_conseq* is used as the default reasoning conclusion.

The other three functions add elements of the data source component to the FSI model given by the parameter *fsi* of these functions. The function *fsi_add_fsa* adds a part of the FSA to the FSI model. It gets as input the LVar and its corresponding *pgeometry* objects annotated by LVals as second and third parameters respectively. The format of the last parameter is the same as the output of the function *spa_creator* (Section 5.3) so that users can directly provide built plateau region objects as input when designing FSI models.

The function *fsi_add_cs* adds the consequent to the FSI model. Its second parameter is the LVar of the consequent (output) of the FSI model in which its LVals are provided as a character vector (*lvals*) and a vector of corresponding membership functions (*mfs*). The consequent has also a domain given by the last parameter (*bounds*).

The last function *fsi_add_rules* adds the fuzzy rules set to the FSI model. These rules are given by a character vector as the second parameter of this function. The definition of a fuzzy rule is user-friendly since users can write it by using the LVars and LVals previously defined and added to the FSI model. We highlight some features when specifying these rules with the *fsr* package. First, the package checks the compatibility between LVars and LVals. Second, the order of the statements in the antecedent is not relevant. Finally, each LVar has to appear at most one time in each fuzzy rule and only one kind of logical connective must be used in the statements of the antecedent. These features aid in avoiding possible contradictions.

Program 4 shows how the aforementioned functions are employed in our running example. The majority of the lines in this program are dedicated to designing the FSI model. This is an expected behavior since the fuzzy inference method requires a set of elements to perform the reasoning conclusions. First, we create an empty FSI model by providing a name and its standard behavior (line 1). The resulting object named *fsi* is based on Mamdani's method since the default values of the parameters are not changed. Next, we add the three parts of the FSA for the LVars *accommodation price* (line 2), *accommodation review* (line 3), and *food safety* (line 4) of our running example. Note that we use the built plateau region objects from Program 2 to add these parts of the FSA. Then we specify the consequent of the FSI model to determine the visiting experience of a specific location. Thus, we define its LVals (line 5) and corresponding membership functions (lines 6-8) that are added to the FSI model (line 9). Note that the visiting experience is a rate between 0 (worst) and 100 (best). Next, we design the fuzzy rules (line 10) and add them to the FSI model (line 11). The rules of our running example express the cost-benefit of staying at a location based on the food safety of restaurants, the accommodation price, and the accommodation review. For instance, the experience is *great* if the user stays in an accommodation with *excellent* reviews and *cut-rate* prices as well as near to restaurants with *high* sanitary inspection grades (last rule in line 10).

The evaluation phase consists of two different functions. The first function implements the spatial reasoning component of FIFUS and has the following signature:

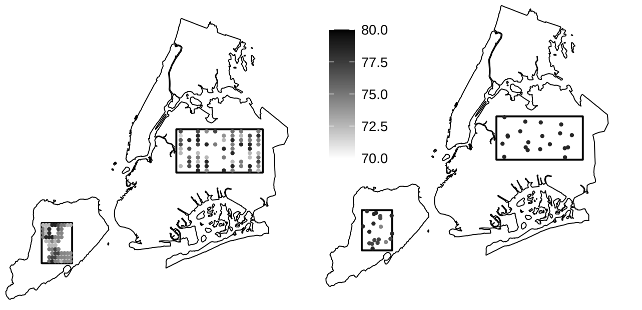
```
fsi_eval(fsi, point, ...)
```

This function evaluates the FSI model indicated by the first parameter by considering a simple point object as the second input parameter. To perform the defuzzification technique, the "three-dots parameter" can be used to inform the function how the elements of the resulting fuzzy set should be discretized. The output of this function is a numeric value that belongs to the domain of the consequent (i.e., as specified by *fsi_add_cs*) and represents the result of the reasoning process.

The second function implements the approaches for evaluating the query window inference and has the following signature:

```
fsi_qw_eval(fsi, qw, approach = "discretization", ...)
```

The first parameter of this function is the built FSI model. The second parameter is an *sfg* object storing the query window that is supposed to be used as input for the inference. The next parameter defines which approach is employed to perform the query window inference. There are two options: *discretization* and *pso*. Each option requires its own set of parameters that are given by using the "three-dots" parameter. For the *discretization* approach, two additional parameters are needed. The first one is the target LVal from the LVar of the consequent. It specifies that only those points with inferred values with membership degrees greater than 0 in the membership function of the target LVal should be included in the final answer of the query window inference. The second parameter is the number k of points that have to be captured from the query window and evaluated by the function *fsi_eval*. To capture them, we create a regular grid inside the query window where the number of columns and rows is the square root of k. For instance, if k is equal to 100, then the grid has 10 rows and 10 columns.



(a) *discretization approach* (b) *pso approach*

Figure 6: The results for the query window inference approaches on two query windows where the larger query window corresponds to the *qw1* of Program 4.

For the *pso* approach, we first discover the user's goal. If the user wants to maximize inferred values, we consider the LVal that contains the greatest values of the domain of the consequent as the target. Otherwise, we consider the LVal with the smallest values of the domain of the consequent since the user wants to minimize the inferred values. For instance, in the running example we aim to know the best locations to visit. Thus, we are interested in capturing the points with inferred values with some degree in the LVal *great*.

Then we split the query window into four subquadrants and apply the PSO algorithm to each subquadrant as its search space. A PSO particle is a simple point object. The fitness function of the PSO employs the function *fsi_eval* to determine the point in the subquadrant with the maximum or minimum inferred value. This means we get four points as a result.

Next, we test if each resulting point belongs to the user's request by checking whether its inferred value has a membership degree in the target LVal. If a point satisfies the user's request, we continue to explore the subquadrant by splitting it again into four subquadrants. This procedure is recursively performed since several points in the query window can satisfy the user's request. The recursion is stopped if a given maximum depth is reached. Hence, it is a needed parameter for the *pso* approach. In addition, the PSO algorithm has its own set of parameters such as the maximum number of iterations and the size of the population (i.e., the number of particles).

Lines 12 to 16 in Program 4 perform the evaluation phase for our FSI model. First, it infers the visiting experience in a single point location (line 12). In lines 13 and 14, we build a query window as an *sfg* object that is the input of the query window inference processed by both approaches (lines 15 and 16). For the *discretization* approach, we set $k = 100$ points and show its result in Figure 6a. For the *pso* approach, we use the maximum depth 2 and show its result in Figure 6b. Further, these figures show the results of these approaches to another query window. Note that the points returned by the *pso* approach are much nearer to the greatest inferred value than the points yielded by the *discretization* approach.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced an R package named *fsr* as an implementation of the fuzzy spatial data types, operations, and predicates of the SPA. The *fsr* package provides a wide range of functions that allow users to (i) construct spatial plateau objects from real datasets, (ii) conduct exploratory (spatial) data analysis by deploying SPA operations, (iii) design FSI models to get new insights from spatial plateau objects, and (iv) optimize the inference process by applying the PSO algorithm. To the best knowledge of the authors, this is the first implementation of a fuzzy spatial type system with support for fuzzy spatial inference that can be used by different types of applications such as SDBS, GIS, and SDS.

Future work will deal with a number of topics. First, we aim to design optimized techniques to implement spatial plateau operations. Second, we plan to conduct an experimental analysis on the query window inference approaches to understand the impact of their parameters on the runtime performance and the quality of their results. Finally, we plan to explore the combination of the best characteristics of both approaches to propose a hybrid approach.

REFERENCES

[1] A. C. Carniel, R. R. Ciferri, and C. D. A. Ciferri. 2016. Handling Fuzzy Points and Fuzzy Lines using the FuzzyGeometry Abstract Data Type. *Journal of Information and Data Management* 7, 1 (2016), 35–51.
[2] A. C. Carniel and M. Schneider. 2016. A Conceptual Model of Fuzzy Topological Relationships for Fuzzy Regions. In *IEEE Int. Conf. on Fuzzy Systems.* 2271–2278.
[3] A. C. Carniel and M. Schneider. 2017. Fuzzy Inference on Fuzzy Spatial Objects (FIFUS) for Spatial Decision Support Systems. In *IEEE Int. Conf. on Fuzzy Systems.* 1–6.
[4] A. C. Carniel and M. Schneider. 2018. Spatial Plateau Algebra: An Executable Type System for Fuzzy Spatial Data Types. In *IEEE Int. Conf. on Fuzzy Systems.* 1–8.
[5] A. C. Carniel and M. Schneider. 2019. A Systematic Approach to Creating Fuzzy Region Objects from Real Spatial Data Sets. In *IEEE Int. Conf. on Fuzzy Systems.* 1–6.
[6] C. Chen, T. R. Razak, and J. M. Garibaldi. 2020. FuzzyR: An Extended Fuzzy Logic Toolbox for the R Programming Language. In *IEEE Int. Conf. on Fuzzy Systems.* 1–8.
[7] S. Davari and N. Ghadiri. 2015. Spatial Database Implementation of Fuzzy Region Connection Calculus for Analysing the Relationship of Diseases. In *Iranian Conf. on Electrical Engineering.* 734–739.
[8] S. Davari and N. Ghadiri. 2019. Fuzzy Region Connection Calculus and Its Application in Fuzzy Spatial Skyline Queries. In *Intelligent Computing.* 659–677.
[9] A. Dilo, P. Bos, P. Kraipeerapun, and R. A. de By. 2006. Storage and Manipulation of Vague Spatial Objects using Existing GIS Functionality. In *Flexible Databases Supporting Imprecision and Uncertainty*, G. Bordogna and G. Psaila (Eds.). Vol. 203. 293–321.
[10] A. Dilo, R. A. de By, and A. Stein. 2007. A System of Types and Operators for Handling Vague Spatial Objects. *Int. Journal of Geographical Information Science* 21, 4 (2007), 397–426.
[11] C.-C. Lee. 1990. Fuzzy Logic in Control Systems: Fuzzy Logic Controller, Part II. *IEEE Trans. on Systems, Man, and Cybernetics* 20, 2 (1990), 419–435.
[12] E. Pebesma. 2018. Simple Features for R: Standardized Support for Spatial Vector Data. *The R Journal* 10, 1 (2018), 439–446.
[13] E. J. Pebesma and R. S. Bivand. 2005. Classes and methods for spatial data in R. *R News* 5, 2 (2005), 9–13.
[14] R. Poli, J. Kennedy, and T. Blackwell. 2007. Particle swarm optimization. *Swarm Intelligence* 1 (2007), 33–57.
[15] M. Schneider. 2014. Spatial Plateau Algebra for Implementing Fuzzy Spatial Objects in Databases and GIS: Spatial Plateau Data Types and Operations. *Applied Soft Computing* 16, 3 (2014), 148–170.
[16] S. Schockaert, M. De Cock, C. Cornelis, and E. E. Kerre. 2008. Fuzzy region connection calculus: Representing vague topological information. *Int. Journal of Approximate Reasoning* 48, 1 (2008), 314–331.
[17] J. Verstraete. 2012. Implementable Representations of Level-2 Fuzzy Regions for Use in Databases and GIS. In *Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems.* 361–370.
[18] J. Verstraete, G. De Tré, and A. Hallez. 2006. Bitmap Based Structures for the Modeling of Fuzzy Entities. *Control and Cybernetics* 35, 1 (2006), 147–164.
[19] H. Wickham and G. Grolemund. 2017. *R for Data Science.* O'Reilly Media.
[20] L. A. Zadeh. 1965. Fuzzy Sets. *Information and Control* 8, 3 (1965), 338–353.
[21] L. A. Zadeh. 1973. Outline of a New Approach to the Analysis of Complex Systems and Decision Processes. *IEEE Trans. on Systems, Man, and Cybernetics* SMC-3, 1 (1973), 28–44.