Chapter 11
A Step Further in the Integration of Data Management and Analysis: Pl/R

Mathieu Basille, Ferdinando Urbano and Joe Conway

Abstract This chapter introduces the Pl/R extension, a very powerful alternative to integrate the features offered by R in the database in a gapless workflow. Pl/R is a loadable procedural language that allows the use of the R engine and libraries directly inside the database, thus embedding R scripts into SQL statements and database functions and triggers. Among many advantages, Pl/R avoids unnecessary data replication, allows the use of a single SQL interface for complex scripts involving R queries and offers a tight integration of data analysis and management processes into the database. In this chapter, you will have a basic overview of the potential of Pl/R for the study of GPS locations. You will be introduced to the use of Pl/R, starting with exercises involving simple calculations in R (logarithms, median and quantiles), followed by more elaborated exercises designed to compute the daylight times of a given location at a given date, or to compute complex home range methods.

Keywords R · Pl/R · Database functions · Statistics

M. Basille (✉)
Fort Lauderdale Research and Education Center, University of Florida,
3205 College Avenue, Fort Lauderdale, FL 33314, USA
e-mail: basille@ase-research.org

F. Urbano
Università Iuav di Venezia, Santa Croce 191 Tolentini, 30135 Venice, Italy
e-mail: ferdi.urbano@gmail.com

J. Conway
credativ LLC, 270 E Douglas Avenue, El Cajon, CA 92020, USA
e-mail: joe.conway@credativ.com

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_11,

Introduction

In Chap. 9, you discovered the importance of a tight integration of management and analysis tools for a proper handling of wildlife tracking data. In Chap. 10, you have seen how R can be connected to the database as a client application to perform advanced analysis algorithms and complex data processing steps. There is a very powerful alternative to integrate the features offered by R and by PostgreSQL/PostGIS in a unique workflow, one that dissolves the boundaries between management and analysis as required by the processing of data from the new generation of wildlife tracking sensors.

This advanced approach is offered by Pl/R[1], a loadable procedural language that enables users to write PostgreSQL functions and triggers in the R programming language. In short, Pl/R integrates R into the database. In fact, it is a PostgreSQL extension that you can install and enable in the database, similarly to how you integrated PostGIS (see Chap. 5). Operationally, this tool allows the use of the R engine and libraries directly inside the database, thus embedding R scripts into SQL statements and database functions. This is to be compared with using R as a client application connected to the database (as in Chap. 10): in this case, data are physically imported into R, where R functions can be run in a dedicated environment. The use of R through Pl/R has therefore many advantages, for example:

- no physical replication of data in the two software programs (i.e. no import/export procedures are needed), thus allowing for better performance and lower memory requirements;
- a single interface (SQL) to access the features offered by both the database and R;
- gapless integration of data analysis and management processes into the database, with the possibility to directly store, manage, and reuse results of analysis to enable meta-analysis.

The integration of R inside the database also opens the door to the automation of real-time analysis performed routinely on massive sets of data. For instance, this gapless framework could be used to set up early warning systems that detect behaviours of the animals that can be potentially dangerous or of particular importance for researchers.

In this chapter, you will be introduced to the use of Pl/R in the context of PostGIS. You will start by exercises involving simple calculations in R (logarithms, median and quantiles) to understand how Pl/R works. More elaborated exercises designed to compute the daylight times of a given location at a given date or to compute complex home range methods will then give you a basic overview of the potential of Pl/R for the study of GPS locations.

[1] See the official website here: http://www.joeconway.com/web/guest/pl/r.

Getting Started with Pl/R

Pl/R, like PostGIS, is an extension of PostgreSQL. The installation procedure is thus similar to PostGIS itself, but will not be covered in this book[2]. However, be sure to have R installed first[3] and that the database user has read access to the directory where R is installed. Once Pl/R is installed, it must be enabled in your database with the command

```
CREATE EXTENSION plr;
```

You can test that it is correctly installed:

```
SELECT * FROM plr_version();
```

Now you can create functions in Pl/R procedural language pretty much the same way you write functions in R. Indeed, the body of a Pl/R function uses the R syntax, because it is actually pure R code! A generic R code snippet such as

```
x <- 10
4/3*pi*x^3
```

can be directly embedded into a Pl/R function in PostgreSQL using a generic function skeleton with the Pl/R language:

```
CREATE OR REPLACE FUNCTION tools.plr_fn ()
RETURNS float8 AS
$BODY$
  x <- 10
  4/3*pi*x^3
$BODY$
LANGUAGE 'plr';
```

The function can then be used in an SQL statement:

```
SELECT tools.plr_fn ();
```

A critical point is to communicate data from the database to and from R. In this simple example, R returns a *numeric* which is recognised by Pl/R as a *float8*. Pl/R can natively handle several types, including booleans (converted to *logical* in R), all forms of integer (converted to *integer*) or numeric (converted to *numeric*) and all forms of text (converted to *character*)[4].

---

[2] See http://www.joeconway.com/plr/doc/plr-install.html for more details.

[3] To download and install R, check your preferred CRAN mirror: http://cran.r-project.org/mirrors.html.

[4] See http://www.joeconway.com/plr/doc/plr-data.html.

You will now start exploring the potential of Pl/R by writing a function *r_log* to calculate the logarithm of a sample of numbers:

```
CREATE OR REPLACE FUNCTION tools.r_log(float8, float8)
RETURNS float AS
$BODY$
  log(arg1, arg2)
$BODY$
LANGUAGE 'plr';
```

Note that functions to compute logarithms already exist in PostgreSQL, so that you can immediately compare the results given by R and PostgreSQL (remember that with a Pl/R function, the R engine does the computation, and PostgreSQL only handles the input and output). In this example, you calculate the natural and the common (base 10) logarithm of the area of the Minimum Convex Polygons (MCP) created in Chap. 9:

```
SELECT
  area, log(area), tools.r_log(area, 10), ln(area), tools.r_log(area, exp(1))
FROM analysis.home_ranges_mcp
WHERE description = 'test all animals at 0.9';
```

The result is

```
  area   |  log   | r_log  |   ln   | r_log
---------+--------+--------+--------+--------
 5.25487 |  0.721 |  0.721 |  1.659 |  1.659
 9.03224 |  0.956 |  0.956 |  2.201 |  2.201
 8.93319 |  0.951 |  0.951 |  2.190 |  2.190
 9.74955 |  0.989 |  0.989 |  2.277 |  2.277
 6.57880 |  0.818 |  0.818 |  1.884 |  1.884
 0.13913 | -0.857 | -0.857 | -1.972 | -1.972
```

Fortunately, the results are consistent whether the logarithms are computed by R or PostgreSQL.

Sample Median and Quantiles

Now, let us go one step further and fill a gap of a missing feature of PostgreSQL, namely the ability to calculate the median, and more generally a given quantile, of a sample. Let us start by the median, which will naturally use the *median* function from R. In this example, you need to pass a sample of values in an array (represented by *float8[]*) to the function *tools.median*:

```
CREATE OR REPLACE FUNCTION tools.median(float8[])
RETURNS float AS
$BODY$
  median(arg1, na.rm = TRUE)
$BODY$
LANGUAGE 'plr';
```

The trick here is that *median* is actually an aggregate function[5] that works on several rows at once. Pl/R provides a set of dedicated support tools[6], such as the *plr_array_accum* function which you will use to write the aggregate function:

```
CREATE AGGREGATE tools.median (float8)
(
  sfunc = plr_array_accum,
  stype = float8[],
  finalfunc = tools.median
);
```

You can test the function on the same set of data used for the previous example, with comparison to the mean:

```
SELECT count(area), avg(area), tools.median(area)
FROM analysis.home_ranges_mcp
WHERE description = 'test all animals at 0.9';
```

The result is

```
 count |  avg   | median
-------+--------+--------
     6 | 6.615 |  7.756
```

One of the most interesting features of aggregate functions is that they can be used on distinct groups as defined by the *GROUP BY* clause. Let us see a working example, which retrieves the average and median elevation for each monitored animal and computes the difference:

```
SELECT
  animals_id, avg(altitude_srtm), tools.median(altitude_srtm), tools.median
  (altitude_srtm) - avg(altitude_srtm) AS diff
FROM main.gps_data_animals
WHERE animals_id != 6 AND gps_validity_code = 1
GROUP BY animals_id
ORDER BY animals_id;
```

The result shows that the median is systematically higher than the mean, which is indicative of a distribution skewed towards low elevations:

```
 animals_id |    avg     | median |  diff
------------+------------+--------+---------
          1 | 1337.101 |   1577 | 239.899
          2 | 1518.995 |   1623 | 104.005
          3 | 1350.491 |   1490 | 139.509
          4 | 1363.569 |   1555 | 191.431
          5 | 1323.017 |   1562 | 238.983
```

[5] http://www.postgresql.org/docs/9.2/static/functions-aggregate.html.

[6] http://www.joeconway.com/plr/doc/plr-pgsql-support-funcs.html.

You will now proceed with the more general quantile function. The approach is slightly more complicated, since the function requires both the sample on which to compute the quantile, and a number to indicate which quantile to compute (between 0 and 1). The aforementioned *plr_array_append* function only works on an array; you will thus first create a new *plr_array_val_append* function to work on an array together with a value (the probability of the quantile), and its associated *array_val* type (note that you store both in the *tools* schema):

```
CREATE TYPE tools.array_val AS (arr float8[], val float8);

CREATE OR REPLACE FUNCTION tools.plr_array_val_append(
  array_val tools.array_val, new_val float8, keep_val float8)
RETURNS tools.array_val CALLED ON NULL INPUT AS
$BODY$
  DECLARE
    arr float8[];
    out record;
  BEGIN
    IF array_val IS NULL THEN
      arr := ARRAY[new_val];
    ELSE
      arr := array_val.arr || new_val;
    END IF;
    out = row(arr, keep_val)::tools.array_val;
    RETURN out;
  END;
$BODY$
LANGUAGE plpgsql;
```

The new *tools.quantile* will now work on a *array_val* object, and the associated aggregate function will use the newly created *tools.plr_array_val_append* function:

```
CREATE OR REPLACE FUNCTION tools.quantile(tools.array_val)
RETURNS float AS
$BODY$
  quantile(unlist(arg1$arr), probs = arg1$val, na.rm = TRUE)
$BODY$
LANGUAGE 'plr';

CREATE AGGREGATE tools.quantile (float8, float8)
(
  sfunc = tools.plr_array_val_append,
  stype = tools.array_val,
  finalfunc = tools.quantile
);
```

You can now try to use the quantile function with different probabilities, and check that the 50 % quantile actually corresponds to the median:

```
SELECT
  count(area), avg(area), tools.median(area), tools.quantile(area, 0.5) AS
  quant50, tools.quantile(area, 0.1) AS quant10, tools.quantile(area, 0.9) AS
  quant90
FROM analysis.home_ranges_mcp
WHERE description = 'test all animals at 0.9';
```

The result is

```
 count |  avg   | median | quant50 | quant10 | quant90
-------+--------+--------+---------+---------+---------
     6 | 6.615 |  7.756 |   7.756 |   2.697 |   9.391
```

Of course, given that you just created an aggregate function, there is no reason not to use the *GROUP BY* clause, for instance to calculate the 5 and 95 % quantiles of the elevation for each animal:

```
SELECT
  animals_id, avg(altitude_srtm), tools.median(altitude_srtm),
  tools.quantile(altitude_srtm, 0.05) AS quant05,
  tools.quantile(altitude_srtm, 0.95) AS quant95
FROM main.gps_data_animals
WHERE animals_id != 6
GROUP BY animals_id
ORDER BY animals_id;
```

This gives the following result:

```
 animals_id |    avg     | median | quant05 | quant95
------------+------------+--------+---------+---------
          1 | 1337.139 |   1577 |     723 |    1679
          2 | 1519.136 |   1623 |    1146 |    1734
          3 | 1350.571 |   1490 |     801 |    1557
          4 | 1363.526 |   1555 |     868 |    1725
          5 | 1323.017 |   1562 |     790 |    1622
```

In the Middle of the Night

One of the most powerful assets of R is its broad and ever-growing package ecosystem (4919 packages at the time of writing[7]). If a statistical method has been developed, it most likely exists for R in a given package. In this example, you are going to implement a useful feature concealed in the *maptools* package, which provides a set of functions able to deal with the position of the sun and compute crepuscule, sunrise and sunset times for a given location at a given date[8]. Although

[7] See the list on CRAN: http://cran.r-project.org/web/packages/available_packages_by_name.html.

[8] This example is based on, and extends, a tutorial from George MacKerron: http://blog.mackerron.com/2012/10/15/sunrise-sunset-postgis-plr/.

the computation of these times depends on the definition you use (e.g. the definition of the horizon, the angle of the sun below or above the horizon), it is beyond the aim of this chapter to enter into details, and you will just use the standard *maptools* approach, which relies on algorithms from the National Oceanic and Atmospheric Administration (NOAA[9]).

For this example, you will need the R packages *rgeos*, *maptools* and *rgdal*: make sure to install them first in R. All these packages will be loaded on demand in the function, but note that Pl/R can also load a list of packages at start-up[10]. As seen earlier, Pl/R can communicate basic data types from PostgreSQL and R, but cannot handle spatial objects. However, both PostgreSQL and R can handle well-known text (WKT) representations, which are simply passed as text strings. The only drawback of this approach is that the standard WKT approach does not include the projection, so that you need to explicitly pass it. Here is the *daylight* function, which returns the sunrise and sunset times (as a text array) for a spatial point expressed as a WKT, with its associated SRID, a timestamp to give the date and a time zone:

```
CREATE OR REPLACE FUNCTION tools.daylight(
  wkt text,
  srid integer,
  datetime timestamptz,
  timezone text)
RETURNS text[] AS
$BODY$
  require(rgeos)
  require(maptools)
  require(rgdal)
  pt <- readWKT(wkt, p4s = CRS(paste0("+init=epsg:", srid)))
  dt <- as.POSIXct(substring(datetime, 1, 19), tz = timezone)
  sr <- sunriset(pt, dateTime = dt, direction = "sunrise",
      POSIXct.out = TRUE)$time
  ss <- sunriset(pt, dateTime = dt, direction = "sunset",
      POSIXct.out = TRUE)$time
  return(c(as.character(sr), as.character(ss)))
$BODY$
LANGUAGE 'plr';
```

Let us try to get the sunrise and sunset times for today, near the municipality of Terlago, northern Italy. Because R and PostgreSQL use different time zone formats, you need to pass the time zone to R literally as '*Europe/Rome*'[11]:

```
SELECT tools.daylight('POINT(11.001 46.001)', 4326, '2012-09-01'
  ::timestamp, 'Europe/Rome');
```

The results indicate a sunrise at 07:26 and a sunset at 18:39, as seen below:

[9] For more details, see: http://www.esrl.noaa.gov/gmd/grad/solcalc/calcdetails.html.

[10] See: http://www.joeconway.com/plr/doc/plr-module-funcs.html.

[11] See *?timezone* in R for more details on the time zone format.

```
                    daylight
-----------------------------------------------
 {"2013-10-10 07:26:08","2013-10-10 18:39:01"}
```

You can now modify this function to return a boolean value (TRUE or FALSE) indicating whether a given time of the day at a given location corresponds to daylight or not. This is the purpose of the *is_daylight* function, which will prove useful to test the daylight for animal locations:

```
CREATE OR REPLACE FUNCTION tools.is_daylight(
  wkt text,
  srid integer,
  datetime timestamptz,
  timezone text)
RETURNS boolean AS
$BODY$
  require(rgeos)
  require(maptools)
  require(rgdal)
  pt <- readWKT(wkt, p4s = CRS(paste0("+init=epsg:", srid)))
  dt <- as.POSIXct(substring(datetime, 1, 19), tz = timezone)
  sr <- sunriset(pt, dateTime = dt, direction = "sunrise",
      POSIXct.out = TRUE)$time
  ss <- sunriset(pt, dateTime = dt, direction = "sunset",
      POSIXct.out = TRUE)$time
  return(ifelse(dt >= sr & dt < ss, TRUE, FALSE))
$BODY$
LANGUAGE 'plr';
```

This function can be used on a single point, e.g. with the same coordinates as above:

```
SELECT tools.is_daylight('POINT(11.001 46.001)', 4326, '2013-10-10
12:34:56'::timestamp, 'Europe/Rome');
```

The result is

```
 is_daylight
-------------
 t
```

Since the function seems to work, you can apply it to GPS locations. Let us run it for the first 10 valid locations:

```
WITH tmp AS (SELECT ('Europe/Rome')::text AS tz)
SELECT
  ST_AsText(geom) AS location,
  acquisition_time AT TIME ZONE tz AS acquisition_time,
  tools.is_daylight(ST_AsText(geom), ST_SRID(geom), acquisition_time
   AT TIME ZONE tz, tz)
FROM main.gps_data_animals, tmp
WHERE gps_validity_code = 1
LIMIT 10;
```

The results directly provide the daylight boolean for each location:

```
          location           |  acquisition_time   | is_daylight
-----------------------------+---------------------+-------------
POINT(11.0484248 46.0126308) | 2005-10-28 02:00:54 | f
POINT(11.0485898 46.0126192) | 2005-10-28 06:00:54 | f
POINT(11.0539 46.0117863)    | 2005-10-28 10:01:53 | t
POINT(11.0526141 46.0101295) | 2005-10-28 18:01:53 | t
POINT(11.0532885 46.006617)  | 2005-10-28 22:01:51 | f
POINT(11.0515082 46.0120316) | 2005-10-29 02:00:54 | f
POINT(11.054181 46.0121311)  | 2005-10-29 06:01:21 | f
POINT(11.0563228 46.0096166) | 2005-10-29 10:01:36 | t
POINT(11.0568896 46.0095597) | 2005-10-29 14:01:53 | t
POINT(11.0554781 46.0089192) | 2005-10-29 18:00:50 | t
```

Extending the Home Range Concept

In Chaps. 5 and 8, the MCP method was introduced, and Chap. 9 described how it can be used to define the notion of home ranges. In this section, you will first reproduce the MCP home ranges, using the *mcp* function from the R package *adehabitatHR*[12]. To do this, you first create a new type *hr* that stores a polygon as a WKT, together with its associated percentage, and the function *mcp_r* to compute the MCP:

```
CREATE TYPE tools.hr AS (percent int, wkt text);

CREATE OR REPLACE FUNCTION tools.mcp_r (wkt text, percent integer)
RETURNS SETOF tools.hr AS
$BODY$
  require(rgeos)
  require(adehabitatHR)
  geom <- readWKT(wkt)
  return(data.frame(percent = percent, wkt = sapply(percent, function(x)
         writeWKT(mcp(geom, x)))))
$BODY$
LANGUAGE plr;
```

The function can be simply called on a collection of points as a WKT and an integer between 0 and 100 (as the percentage of locations kept for the computation):

```
SELECT (tools.mcp_r(ST_AsText(ST_Collect(geom)), 90)).*
FROM main.gps_data_animals
WHERE gps_validity_code = 1 AND animals_id = 1;
```

[12] http://cran.r-project.org/web/packages/adehabitatHR/.

The result is thus a combination of the percentage and the WKT representation of the MCP:

```
 percent |                            wkt
---------+-------------------------------------------------------------
      90 | POLYGON ((11.0822929999999999 46.0084694000000027, [...]
```

To make sure that the function works correctly, you can compare the outputs with the home ranges created in Chap. 9 and stored in *analysis.home_ranges_mcp*, using an MCP with 90 % of the relocations:

```
WITH
  mcpr AS (
    SELECT
      animals_id,(tools.mcp_r(ST_AsText(ST_Collect(geom)), 90)).*
    FROM main.gps_data_animals
    WHERE gps_validity_code = 1 AND animals_id <> 6
    GROUP BY animals_id)
SELECT
  mcpr.animals_id, mcpr.percent,
  ST_Area(geography(wkt)) / 1000000 AS area_r, mcp.area AS area_pg
FROM mcpr, analysis.home_ranges_mcp AS mcp
WHERE mcpr.animals_id = mcp.animals_id
 AND mcp.description = 'test all animals at 0.9'
GROUP BY mcpr.animals_id, mcpr.wkt, mcp.area, percent
ORDER BY mcpr.animals_id;
```

As you can see in the following results, the computations are very similar and only slight discrepancies are visible, caused by using different approaches in selecting a given percentage of locations to compute the MCP:

```
 animals_id | percent | area_r | area_pg
------------+---------+--------+---------
          1 |      90 |  5.255 |   5.255
          2 |      90 |  9.032 |   9.032
          3 |      90 |  8.933 |   8.933
          4 |      90 |  9.749 |   9.750
          5 |      90 |  6.578 |   6.579
```

Let us now introduce a different approach of defining a home range. Instead of a mere polygon, a home range can be defined by the probability that an animal is found at a given point, which is called a utilisation distribution (UD). The core areas of the home range, which are used more often, are then associated with a higher probability; as a consequence, it is also possible to derive the polygon that corresponds to the minimum area in which an animal has a given probability of being located. The simplest UD approach relies on the kernel method, which basically applies a bivariate normal distribution around each location and sums these distribution over the landscape. As no function in PostGIS enables the computation of kernel home ranges, you will wrap the *kernelUD* function from *adehabitatHR* into a new function *kernelud*, following an approach very similar to the *mcp_r* function:

```
CREATE OR REPLACE FUNCTION tools.kernelud (wkt text, percent integer)
RETURNS SETOF tools.hr AS
$BODY$
  require(rgeos)
  require(adehabitatHR)
  geom <- readWKT(wkt)
  kud <- kernelUD(geom)
  return(data.frame(percent = percent, wkt = sapply(percent, function(x)
         writeWKT(getverticeshr(kud, x)))))
$BODY$
LANGUAGE plr;
```

You can thus query the table with all animal locations to compute the kernel home range, for instance for animal 1 at 50, 90, and 95 %:

```
WITH tmp AS (SELECT unnest(ARRAY[50,90,95]) AS pc)
SELECT (tools.kernelud(ST_AsText(ST_Collect(geom)), pc)).*
FROM main.gps_data_animals, tmp
WHERE gps_validity_code = 1 AND animals_id = 1
GROUP BY pc
ORDER BY pc;
```

The result is a list of *hr* objects:

```
 percent |                                  wkt
---------+-------------------------------------------------------------------
      50 | MULTIPOLYGON (((11.0383491399999993 46.0039599699999968, [...]
      90 | MULTIPOLYGON (((11.0314047899999998 46.0035943099999969, [...]
      95 | MULTIPOLYGON (((11.0314047899999998 46.0009246899999980, [...]
```

You will now create a table *analysis.home_ranges_kernelud* to store the different kernel home ranges, exactly as the *analysis.home_ranges_mcp* stores the MCP home ranges:

```
CREATE TABLE analysis.home_ranges_kernelud(
  home_ranges_kernelud_id serial NOT NULL,
  animals_id integer NOT NULL,
  start_time timestamp with time zone NOT NULL,
  end_time timestamp with time zone NOT NULL,
  num_locations integer,
  area numeric(13,5),
  geom geometry (multipolygon, 4326),
  percentage double precision,
  insert_timestamp timestamp with time zone
    DEFAULT now(),
  CONSTRAINT home_ranges_kernelud_pk
    PRIMARY KEY (home_ranges_kernelud_id),
  CONSTRAINT home_ranges_kernelud_animals_fk
    FOREIGN KEY (animals_id)
    REFERENCES main.animals (animals_id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION);
COMMENT ON TABLE analysis.home_ranges_kernelud
```

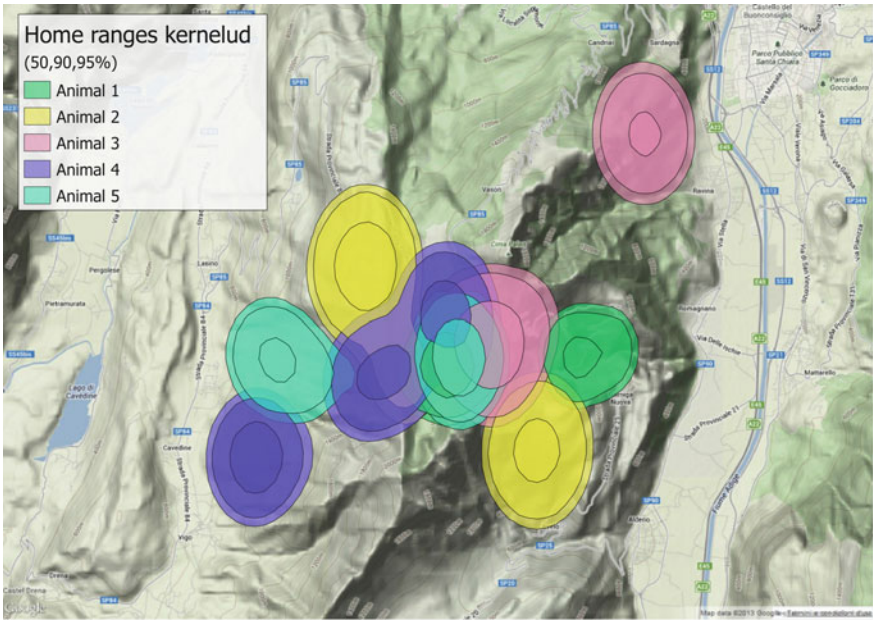

Fig. 11.1 Kernel home ranges at 50, 90 and 95 %

```
IS 'Table that stores the home range polygons derived from kernelUD. The
area is computed in squared km.';

CREATE INDEX fki_home_ranges_kernelud_animals_fk
  ON analysis.home_ranges_kernelud
  USING btree (animals_id);
CREATE INDEX gist_home_ranges_kernelud_index
  ON analysis.home_ranges_kernelud
  USING gist (geom);
```

Let us now populate this table using 50 and 90 % kernels for all animals (see the graphical results in Fig. 11.1):

```
WITH
  tmp AS (SELECT unnest(ARRAY[50,90,95]) AS pc),
  kud AS (
    SELECT
      animals_id,
      min(acquisition_time) AS start_time,
      max(acquisition_time) AS end_time,
      count(animals_id) AS num_locations,
      (tool.kernelud(ST_AsText(ST_Collect(geom)), pc)).*
    FROM main.gps_data_animals, tmp
```

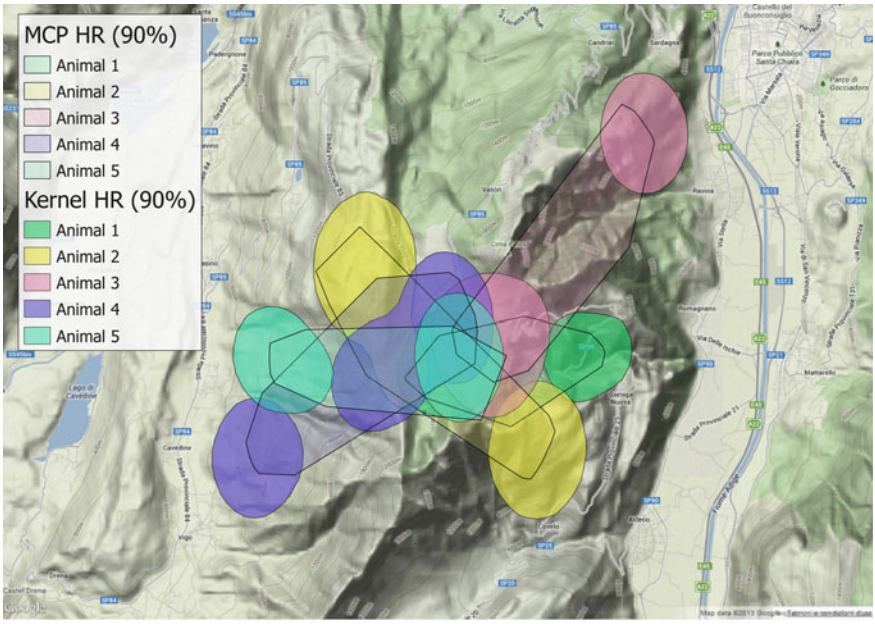

Fig. 11.2 Comparison between kernel and MCP home range at 90 %

```
    WHERE
      gps_validity_code = 1 AND animals_id <> 6
    GROUP BY animals_id,pc
    ORDER BY animals_id,pc)
INSERT INTO analysis.home_ranges_kernelud (animals_id, start_time, end_time,
  num_locations, area, geom, percentage)
SELECT
  animals_id,
  start_time,
  end_time,
  num_locations,
  ST_Area(geography(wkt)) / 1000000,
  ST_GeomFromText(wkt, 4326),
  percent / 100.0
FROM kud
ORDER BY animals_id, percent;
```

You can now compare the outputs from the MCP and the kernel home ranges. You thus retrieve the results from the MCP and the kernel table, using the home ranges estimated at 90 %. For each animal, you also compute the area of the home range overlap as estimated by both methods (using *ST_Intersection* to define the shared area), and the proportion of common area (using *ST_Union*) that it represents:

```
SELECT
  mcp.animals_id AS ani_id,
  mcp.area AS mcp_area,
  kud.area AS kud_area,
  ST_Area(geography(ST_Intersection(mcp.geom, kud.geom))) / 1000000 AS
  overlap,
  ST_Area(geography(ST_Intersection(mcp.geom, kud.geom))) /
  ST_Area(geography(ST_Union(mcp.geom, kud.geom))) AS over_prop
FROM
  analysis.home_ranges_mcp AS mcp,
  analysis.home_ranges_kernelud AS kud
WHERE
  mcp.animals_id = kud.animals_id AND
  mcp.percentage = kud.percentage AND
  mcp.percentage = 0.9;
```

Note that the percentage in each function is not exactly the same, which should prevent any conclusion from the comparison: for the MCP, it relates to the proportion of locations used in the computation, while for the kernel, it relates to the density of the UD. Nevertheless, they provide polygons with very similar areas, which is surprising! But, as you can see from the proportion of overlap, and in Fig. 11.2, the areas depicted by both methods are actually very different and highlight the different philosophies underlying each method:

```
 ani_id | mcp_area | kud_area | overlap | over_prop
--------+----------+----------+---------+-----------
      1 |    5.255 |    5.352 |   3.054 |     0.404
      2 |    9.032 |    9.880 |   5.516 |     0.412
      3 |    8.933 |    8.090 |   3.972 |     0.304
      4 |    9.750 |    9.869 |   6.586 |     0.505
      5 |    6.579 |    6.344 |   3.608 |     0.387
```

As a final note, beware that projections were purposely ignored in this exercise. In particular, the *kernulUD* function from *adehabitatHR* assumes that you are using planar coordinates (from a Cartesian coordinate system such as UTM), but not geographic coordinates (longitude, lattitude), and does not check for it. Indeed, using geographic coordinates could result in inaccurate results because they are processed as planar coordinates. More accurate results would be achieved by first reprojecting the data in a planar coordinate system, e.g. in UTM, and converting the results back into geographic coordinates. Here is such an example on animal 1, using the *tools.srid_utm* function, presented in Chap. 9, that calculates the SRID of the UTM zone where the centroid of the data set is located:

```
WITH
  srid AS (
    SELECT tools.srid_utm(
      ST_X(ST_Centroid(ST_Collect(geom))),
      ST_Y(ST_Centroid(ST_Collect(geom)))) AS utm
    FROM main.gps_data_animals
    WHERE gps_validity_code = 1 AND animals_id = 1),
  kver AS (
    SELECT (tools.kernelud(ST_AsText(
      ST_Transform(ST_Collect(geom), srid.utm)), 90)).*
    FROM main.gps_data_animals, srid
    WHERE gps_validity_code = 1 AND animals_id = 1
    GROUP BY srid.utm)

SELECT
  kver.percent AS pc,
  ST_AsEWKT(
    ST_Transform(
      ST_GeomFromText(kver.wkt, srid.utm),
      4326)) AS ewkt
FROM kver, srid;
```

This gives the following result:

```
 pc |                            ewkt
----+-------------------------------------------------------------------
 90 | SRID=4326;MULTIPOLYGON(((11.0312843826874 46.0048190563777, [...]
```

Conclusions and Perspectives

In this chapter, you only briefly tackled the possibilities of Pl/R. In Chap. 10, you were presented an extensive overview of the use of R in the field of animal ecology, and how R can nicely complement PostGIS for the study of animal locations. However, you saw that the flow between PostGIS and R is not always linear: it is sometimes required to send data from R back to PostGIS, run some further spatial queries and retrieve the results again in R. PostGIS offers some very useful features that R does not, such as the online publication and interactive mapping of spatial data[13]. Lastly, it might be necessary to use only one language in order to evaluate complex scripts. For all these reasons, the potential of Pl/R for the biologist is immense, but you have barely scratched the surface of the possibilities here and the development of Pl/R in the context of spatial data will likely grow in the coming years.

As you could see in the few examples provided in this chapter, the main challenge in using Pl/R is to communicate data from PostGIS to R, and back. While Pl/R can only handle basic data types (all kinds of numeric, text and boolean), it cannot directly handle spatial and temporal objects. Fortunately, you

[13] See for instance MapServer: http://mapserver.org/.

saw that the WKT representation could be used to this end and allows you to handle vector features (points, lines and polygons) in a straightforward manner. Other possibilities exist too. For simple cases, you could also pass directly spatial coordinates, using for instance *ST_X(geom)* or *ST_Y(geom)*, and passing them to R as numbers, which would then be converted to spatial objects in R (exactly as was done in Chap. 10, but wrapped in a Pl/R function). This is perfectly valid for simple geometries (i.e. a set of points or segments) for which it is easy to manually handle the coordinates, but more complex geometries (a collection of multilines or multipolygons) would rather quickly become intractable, in which case the WKT approach offers a robust and flexible standard approach. Pl/R also offers a function (*pg.spi.exec*) to directly evaluate SQL code from the body of the function, which can be very useful in some cases, especially when the data would be too complicated to pass in the arguments (in essence, it is similar to the *dbGetQuery* function from the R package *RpostgreSQL*).

The last two things to consider involve the most complex data types. First of all, Pl/R is also able to handle binary data types (*bytea* objects). This can be very useful in many cases, when the object of interest computed in R has no correspondence to PostgreSQL objects, but you still would like to store it in the database. Imagine, for instance, a *ltraj* object (see Chap. 10), or a PNG figure that you would like to communicate to a Web server for display in a browser[14]. It would be immensely complex to convert these objects using PostgreSQL data types. However, using *bytea* objects allows you to store them when necessary, and to use them again in a software able to deal with them (e.g. R for *ltraj* objects or any Web server for an image). Finally, there was no example in this chapter dealing with rasters, because there is no simple way to deal with them in a Pl/R function. At the moment of writing, there is no way to pass a raster in the arguments, as WKT representations of rasters are not standardised yet. Possible solutions involve the use of the package *rgdal* (i.e. *readGDAL* to import a raster to R, and *writeGDAL* to send it back to the database), or directly *raster2pgsql* in a system call to write rasters into the database[15]. Unfortunately, both approaches require you to pass credentials to access the database as arguments, or, worse, to directly include them in the function (which is definitely not a good practice). However, progress in this area can only improve the situation in the coming years.

[14] See an example here: http://www.joeconway.com/web/guest/pl/r/-/wiki/Main/Bytea+Graphing+Example.

[15] Another solution might be to use the TerraLib library, which involves another set of dependencies: http://www.terralib.org/.