Chapter 9
Exploring Tracking Data: Representations, Methods and Tools in a Spatial Database

Ferdinando Urbano, Mathieu Basille and Pierre Racine

Abstract The objects of movement ecology studies are animals whose movements are usually sampled at more-or-less regular intervals. This spatiotemporal sequence of locations is the basic, measured information that is stored in the database. Starting from this data set, animal movements can be analysed (and visualised) using a large set of different methods and approaches. These include (but are not limited to) trajectories, raster surfaces of probability density, points, (home range) polygons and tabular statistics. Each of these methods is a different representation of the original data set that takes into account specific aspects of the animals' movement. The database must be able to support these multiple representations of tracking data. In this chapter, a wide set of methods for implementing many GPS tracking data representations into a spatial database (i.e. with SQL code and database functions) are introduced. The code presented is based on the database created in Chaps. 2, 3, 4, 5, 6, 7 and 8.

Keywords Animal movement · Trajectory · Home range · Database functions · Movement parameters

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

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

P. Racine
Centre for Forest Research, University Laval, Pavillon Abitibi-Price, 2405 de la Terrasse, Bureau 1122, Quebec City, QC G1V 0A6, Canada
e-mail: pierre.racine@sbf.ulaval.ca

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_9,

Introduction

The objects of movement ecology studies are animals whose movements are usually sampled at more-or-less regular intervals. This spatiotemporal sequence of locations is the basic, measured information that is stored in the database. Starting from this data set, animal movements can be analysed (and visualised) using a large set of different methods and approaches. These include (but are not limited to) trajectories, raster surfaces of probability density, points, (home range) polygons and tabular statistics. Each of these methods is a different representation of the original data set that takes into account specific aspects of the animals' movement. The database must be able to support these multiple representations of tracking data.

Although some very specific algorithms (e.g. kernel home range) must be run in a dedicated GIS or spatial statistics environment (see Chaps. 10 and 11), a number of analyses can be implemented directly in PostgreSQL/PostGIS. This is possible due to the large set of existing spatial functions offered by PostGIS and to the powerful but still simple possibility of combining and customising these tools with procedural languages for applications specific to wildlife tracking. What makes the use of databases to process tracking data very attractive is that databases are specifically designed to perform a massive number of simple operations on large data sets. In the recent past, biologists typically undertook movement ecology studies in a 'data poor, theory rich' environment, but in recent years this has changed as a result of advances in data collection techniques. In fact, in the case of GPS data, for which the sampling interval is usually frequent enough to provide quite a complete picture of the animal movement, the problem is not to derive new information using complex algorithms run on limited data sets (as for VHF or Argos Doppler data), but on the contrary to synthesise the huge amount of information embedded in existing data in a reduced set of parameters.

Complex models based on advanced statistical tools are still important, but the focus is on simple operations performed in near real time on a massive data flow. Databases can support this approach, giving scientists the ability to test their hypotheses or provide managers the compact set of information that they need to take their decisions. The database can also be used in connection with GIS and spatial statistical software. The database can preprocess data in order to provide more advanced algorithms the data set requires for the analysis. In the exercise for this chapter, you will create a number of functions[1] to manipulate and prepare data for more complex analysis. These include functions to extract environmental

[1] The concepts behind many of the tools presented in this chapter derive from the work of Clement Calenge for Adehabitat (cran.r-project.org/web/packages/adehabitat/index.html), an R package for tracking data analysis.

statistics from a set of GPS positions; create (and store) trajectories; regularise trajectories (subsample and spatially interpolate GPS positions at a defined time interval); define bursts; compute geometric parameters (e.g. spatial and temporal distance between GPS positions, relative and absolute angles, speed); calculate home ranges based on a minimum convex polygon (MCP) algorithm; and run and store analyses on trajectories. These are examples that can be used to develop your own tools.

Extraction of Statistics from the GPS Data Set

A first, simple example of animal movement modelling and representation based on GPS positions is the extraction of statistics to characterise animals' environmental preferences (in this case, minimum, maximum, average and standard deviation of altitude, and the number of available GPS positions):

```
SELECT
  animals_id,
  min(altitude_srtm)::integer AS min_alt,
  max(altitude_srtm)::integer AS max_alt,
  avg(altitude_srtm)::integer AS avg_alt,
  stddev(altitude_srtm)::integer AS alt_stddev,
  count(*) AS num_loc
FROM main.gps_data_animals
WHERE gps_validity_code = 1
GROUP BY animals_id
ORDER BY avg(altitude_srtm);
```

The result is

```
 animals_id | min_alt | max_alt | avg_alt | alt_stddev | num_loc
------------+---------+---------+---------+------------+---------
          6 |     678 |     989 |     774 |         58 |     278
          5 |     596 |    1905 |    1323 |        337 |    2695
          1 |     686 |    1816 |    1337 |        356 |    1647
          3 |     588 |    1567 |    1350 |        257 |    1826
          4 |     688 |    1887 |    1364 |        332 |    2641
          2 |     926 |    1816 |    1519 |        206 |    2194
```

It is also possible to calculate similar statistics for categorised attributes, like land cover classes:

```
SELECT
  animals_id, (count(*)/tot::double precision)::numeric(5,4) AS percentage,
  label1
FROM
  main.gps_data_animals,
  env_data.corine_land_cover_legend,
  (SELECT animals_id AS x, count(*) AS tot
   FROM main.gps_data_animals
   WHERE gps_validity_code = 1
   GROUP BY animals_id) a
WHERE
  gps_validity_code = 1 AND
  animals_id = x AND
  corine_land_cover_code = grid_code
GROUP BY animals_id, label1, tot
ORDER BY animals_id, label1;
```

The result is

```
 animals_id | percentage |            label1
------------+------------+------------------------------
          1 |     0.3036 | Agricultural areas
          1 |     0.6964 | Forest and semi natural areas
          2 |     0.0251 | Agricultural areas
          2 |     0.9749 | Forest and semi natural areas
          3 |     0.4578 | Agricultural areas
          3 |     0.5422 | Forest and semi natural areas
          4 |     0.3268 | Agricultural areas
          4 |     0.6732 | Forest and semi natural areas
          5 |     0.3662 | Agricultural areas
          5 |     0.6338 | Forest and semi natural areas
          6 |     0.5791 | Agricultural areas
          6 |     0.0108 | Artificial surfaces
          6 |     0.4101 | Forest and semi natural areas
```

A New Data Type for GPS Tracking Data

Before adding new tools to your database, it is useful to define a new composite data type[2]. The new data type combines the simple set of attributes *animals_id* (as *integer*), *acquisition_time* (as *timestamp with time zone*), and *geom* (as *geometry*) and can be used by most of the functions that can be developed for tracking data. Having this data type, it becomes easier to write functions to process GPS locations. First create a data type that combines these attributes:

[2] http://www.postgresql.org/docs/9.2/static/sql-createtype.html.

```
CREATE TYPE tools.locations_set AS (
  animals_id integer,
  acquisition_time timestamp with time zone,
  geom geometry(point, 4326));
```

You can also create a view where this subset of information is retrieved from *gps_data_animals*:

```
CREATE OR REPLACE VIEW main.view_locations_set AS
  SELECT
    gps_data_animals.animals_id,
    gps_data_animals.acquisition_time,
    CASE
      WHEN gps_data_animals.gps_validity_code = 1 THEN
        gps_data_animals.geom
      ELSE NULL::geometry
    END AS geom
  FROM main.gps_data_animals
  WHERE gps_data_animals.gps_validity_code != 21
  ORDER BY gps_data_animals.animals_id, gps_data_animals.acquisition_time;
COMMENT ON VIEW main.view_locations_set
IS 'View that stores the core information of the set of GPS positions (id of
the animal, the acquisition time and the geometry), where non valid records
are represented with empty geometry.';
```

The result is the complete set of GPS locations stored in *main.gps_data_animals* with a limited set of attributes. As you can see, for locations without valid coordinates (*gps_validity_code ! = 1*), the geometry is set to *NULL*. Records with duplicated acquisition times are excluded from the data set. This view can be used as a reference for the functions that have to deal with the *locations_set* data set.

Representations of Trajectories

You can exploit the *locations_set* data type to create trajectories and permanently store them in a table. For a general introduction to trajectories in wildlife ecology, see Calenge et al. (2009), which is also a major reference for a review of the possible approaches in wildlife tracking data analysis. First, you can create the table to accommodate trajectories (see the comment in the function itself for more details):

```
CREATE TABLE analysis.trajectories (
  trajectories_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,
  description character varying, ref_user character varying,
  num_locations integer, length_2d integer,
  insert_timestamp timestamp with time zone DEFAULT now(),
  original_data_set character varying,geom geometry(linestring, 4326),
  CONSTRAINT trajectories_pk
    PRIMARY KEY (trajectories_id),
  CONSTRAINT trajectories_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.trajectories
IS 'Table that stores the trajectories derived from a set of selected
locations. Each trajectory is related to a single animal. This table is
populated by the function tools.make_traj. Each element is described by a
number of attributes: the starting date and the ending date of the location
set, a general description (that can be used to tag each record with specific
identifiers), the user who did the analysis, the number of locations (or
vertex of the lines) that produced the analysis, the length of the line, and
the SQL that generated the dataset.';
```

Then, you can create a function that produces the trajectories and stores them in the table *analysis.trajectories*. This function creates a trajectory given an SQL code that selects a set of GPS locations (as *locations_set* object) where users can specify the desired criteria (e.g. id of the animal, start and end time). It is also possible to add a second parameter: a text that is used to comment the trajectory. A trajectory will be created for each animal in the data set.

```
CREATE OR REPLACE FUNCTION tools.make_traj (
  locations_set_query character varying DEFAULT
  'main.view_locations_set'::character varying,
  description character varying DEFAULT 'Standard trajectory'::character
  varying)
RETURNS integer AS
$BODY$
DECLARE
  locations_set_query_string character varying;
BEGIN
  locations_set_query_string = (SELECT replace(locations_set_query,'''',''''''));
  EXECUTE
    'INSERT INTO analysis.trajectories (animals_id, start_time, end_time,
    description, ref_user, num_locations, length_2d, original_data_set, geom)
    SELECT sel_subquery.animals_id, min(acquisition_time),
    max(acquisition_time), ''' ||description|| ''', current_user, count(*),
    ST_length2d_spheroid(ST_MakeLine(sel_subquery.geom),
    ''SPHEROID("WGS84",6378137,298.257223563)''::spheroid), '''||
```

```
    locations_set_query_string ||''', ST_MakeLine(sel_subquery.geom) AS geom
     FROM
       (SELECT *
        FROM ('||locations_set_query||') a
        WHERE a.geom IS NOT NULL
        ORDER BY a.animals_id, a.acquisition_time) sel_subquery
        GROUP BY sel_subquery.animals_id;';
        raise notice 'Operation correctly performed. Record inserted into
        analysis.trajectories';
  RETURN 1;
END;
$BODY$
LANGUAGE plpgsql;

COMMENT ON FUNCTION tools.make_traj(character varying, character varying) IS
'This function produces a trajectory from a locations_set object (animals_id,
acquisition_time, geom) in the table analysis.trajectories. Two parameters are
accepted: the first is the SQL code that generates the locations_set object,
the second is a string that is used to comment the trajectory. A trajectory
will be created for each animal in the data set and stored as a new record in
the table. If you need to include a single quote in the SQL that selects the
locations (for example, when you want to define a timestamp), you have to use
an additional single quote to escape it.';
```

Note that in PostgreSQL, if you want to add a single quote in a string (‘), which is usually the character that closes a string, you have to use an escape character before[3]. This can be done using two single quotes (‘‘): the result in the string will be a single quote. Here are two examples of use. The first example is

```
SELECT
  tools.make_traj(
    'SELECT * FROM main.view_locations_set WHERE acquisition_time > ''2006-
    01-01''::timestamp AND animals_id = 3', 'First test');
```

The second example is

```
SELECT
  tools.make_traj(
    'SELECT animals_id, acquisition_time, geom FROM main.gps_data_animals
    WHERE gps_validity_code = 1 AND acquisition_time <''2006-01-01''::
    timestamp', 'Second test');
```

The outputs are stored in the *analysis.trajectories* table. You can see the results in tabular format with

```
SELECT * from analysis.trajectories;
```

[3] http://www.postgresql.org/docs/9.2/static/sql-syntax-lexical.html.

A subset of the fields returned from this query is reported below.

```
 trajectories_id | animals_id | description | num_locations | length_2d
-----------------+------------+-------------+---------------+-----------
               1 |          3 | First test  |          1426 |    288928
               2 |          1 | Second test |           332 |     70043
               3 |          2 | Second test |          1614 |    307836
               4 |          3 | Second test |           400 |     73284
               5 |          4 | Second test |           424 |     72499
               6 |          6 | Second test |           278 |     40602
```

You can compare the length calculated on a 2D trajectory and on a 3D trajectory (i.e. also considering the vertical displacement). This is the code for the 2D trajectory:

```
SELECT
  sel_subquery.animals_id,
  ST_length(
    ST_MakeLine(sel_subquery.geom)::geography)::integer AS lengt_line2d,
  ST_numpoints(
    ST_MakeLine(sel_subquery.geom)) AS num_locations
FROM
  (SELECT
    gps_data_animals.animals_id,
    gps_data_animals.geom,
    gps_data_animals.acquisition_time
   FROM main.gps_data_animals
   WHERE gps_validity_code = 1
   ORDER BY gps_data_animals.animals_id, gps_data_animals.acquisition_time)
    sel_subquery
GROUP BY
  sel_subquery.animals_id;
```

The result is

```
 animals_id | length_line2d | num_locations
------------+---------------+---------------
          1 |        287284 |          1647
          2 |        433959 |          2194
          3 |        362232 |          1826
          4 |        480911 |          2641
          5 |        628674 |          2695
          6 |         40602 |           278
```

This is the code for the 3D trajectory:

```
SELECT
  animals_id,
  ST_3DLength_Spheroid(
    ST_SetSrid(ST_MakeLine(geom), 4326),
    'SPHEROID("WGS84",6378137,298.257223563)'::spheroid)::integer AS
    length_line3d,
  ST_NumPoints(ST_SetSrid(ST_MakeLine(geom), 4326)) AS num_locations
FROM
  (SELECT
    gps_data_animals.animals_id,
    gps_data_animals.acquisition_time,
    ST_SetSRID(ST_makepoint(
      ST_X(gps_data_animals.geom),
      ST_Y(gps_data_animals.geom),
      gps_data_animals.altitude_srtm::double precision,
      date_part('epoch'::text, gps_data_animals.acquisition_time)), 4326)
      AS geom
   FROM main.gps_data_animals
   WHERE gps_validity_code = 1
   ORDER BY gps_data_animals.animals_id, gps_data_animals.acquisition_time) a
GROUP BY animals_id;
```

The result is

```
 animals_id | length_line3d | num_locations
------------+---------------+---------------
          1 |        296676 |          1647
          2 |        448134 |          2194
          3 |        374117 |          1826
          4 |        491886 |          2641
          5 |        639680 |          2695
          6 |         43372 |           278
```

You can see how in an alpine environment the difference can be relevant. Many functions in PostGIS support 3D objects. For a complete list, you can check the documentation[4]. You can also store points as 3DM objects, where not just the altitude is considered, but also a measure that can be associated with each point. For tracking data, this can be used to store, embedded in the spatial attribute, the acquisition time. As the timestamp data type cannot be used directly, it can be transformed to an integer using *epoch*[5], an integer that represents the number of seconds since 1 January 1970.

[4] http://www.postgis.org/docs/PostGIS_Special_Functions_Index.html#PostGIS_3D_Functions.

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

Regularisation of GPS Location Data Sets

Another useful tool is the regularisation of the location data set. Many times the acquisition time of the GPS sensor is scheduled at a varying frequency. The function introduced below transforms an irregular time series into a regular one, i.e. with a fixed time step. Records that do not correspond to the desired frequency are discharged, while if no record exists at the required time interval, a (virtual) record with the timestamp but no coordinates is created (see the comments embedded in the function for more information on input parameters). Note that this function does not perform any interpolation, but simply resamples the available locations adding a record with NULL coordinates where necessary.

```
CREATE OR REPLACE FUNCTION tools.regularize(
  animal integer,
  time_interval integer DEFAULT 10800,
  buffer double precision DEFAULT 600,
  starting_time timestamp with time zone DEFAULT NULL::timestamp with time zone,
  ending_time timestamp with time zone DEFAULT NULL::timestamp with time zone)
RETURNS SETOF tools.locations_set AS
$BODY$
DECLARE
  location_set tools.locations_set%rowtype;
  cursor_var record;
  interval_length integer;
  check_animal boolean;
BEGIN
-- Error trapping: if the buffer is > 0.5 * time interval, I could take 2
times the same locations, therefore an exception is raised
IF buffer > 0.5 * time_interval THEN
  RAISE EXCEPTION 'With a buffer (%) > 0.5 * time interval (%), you could get
  twice the same location, please reduce buffer or increase time interval.',
  buffer, time_interval;
END IF;
-- If the starting date is not set, the minimum, valid timestamp of the data
set is taken
IF starting_time IS NULL THEN
  SELECT
    min(acquisition_time)
  FROM
    main.view_locations_set
  WHERE
    view_locations_set.animals_id = animal
  INTO starting_time;
END IF;
-- If the ending date is not set, the maximum, valid timestamp of the data set
is taken
IF ending_time IS NULL THEN
  SELECT max(acquisition_time)
```

```
  FROM main.view_locations_set
  WHERE view_locations_set.animals_id = animal
  INTO ending_time;
END IF;

-- I define the interval time (number of seconds between the starting and
ending time)
SELECT extract(epoch FROM (ending_time-starting_time))::integer + buffer
INTO interval_length;

-- I create a 'virtual' set of records with regular time intervals (from
starting_time to ending_time, with a step equal to the interval length; then I
go through all the elements of the virtual set and check whether a real record
exists in main.view_locations_set that has an acquisition_time closer then the
defined buffer. If more then 1 record exists in the buffer range, then I take
the 'closest'.
FOR location_set IN
  SELECT
    animal,
    (starting_time + generate_series (0, interval_length, time_interval) *
    interval '1 second'),
    NULL::geometry
LOOP
  SELECT geom, acquisition_time
  FROM main.view_locations_set
  WHERE
    animals_id = animal AND
    (acquisition_time < (location_set.acquisition_time + interval '1 second'
    * buffer) AND
    acquisition_time > (location_set.acquisition_time - interval '1 second'
    * buffer))
  ORDER BY
    abs(extract (epoch FROM (acquisition_time - location_set.acquisition_time)))
  LIMIT 1
  INTO cursor_var;

-- If I have a record in main.view_locations_set, I get the values from
there, otherwise I keep my 'virtual' record
  IF cursor_var.acquisition_time IS NOT NULL THEN
    location_set.acquisition_time = cursor_var.acquisition_time;
    location_set.geom = cursor_var.geom;
  END IF;
  RETURN NEXT location_set;
END LOOP;
RETURN;
END;
$BODY$
LANGUAGE plpgsql;
```

```
COMMENT ON FUNCTION tools.regularize(integer, integer, double precision,
timestamp with time zone, timestamp with time zone)
IS 'This function creates a complete, regular time series of locations from
main.view_locations_set using an individual id, a time interval (in
seconds), a buffer time (in seconds, which corresponds to the  accepted
delay of GPS recording), a starting time (if no values is defined, the first
record of the animal data set is taken), and an ending time (if no value is
defined, the last record of the animal data set is taken). The function
checks at every time step whether a real record (with or without coordinates)
in the main.view_locations_set table exists (which is the
locations_set object of the "main.gps_data_animals table): if any real data
exist (inside a defined time interval buffer from the reference timestamp
generated by the function) in main.view_locations_set, the real record is
used, otherwise a virtual record is created (with empty geometry). The
output is a table with the structure "location_set" (animals_id integer,
acquisition_time timestamp with time zone, geom geometry).';
```

You can test the effects of the function, comparing the different results with the original data set. For instance, let us extract a regular trajectory for animal 6 with a time interval of 8 h (i.e. $60 \times 60 \times 8$ s):

```
SELECT animals_id, acquisition_time, ST_AsText(geom)
FROM tools.regularize(6, 60*60*8)
LIMIT 15;
```

The first 15 results (out of a total of 96) are

```
 animals_id |    acquisition_time    |          st_astext
------------+------------------------+-----------------------------
          6 | 2005-04-04 08:01:41+02 | POINT(11.0633742 46.0649085)
          6 | 2005-04-04 16:03:08+02 | POINT(11.0626891 46.0651272)
          6 | 2005-04-05 00:03:07+02 |
          6 | 2005-04-05 08:01:50+02 | POINT(11.063423 46.0648249)
          6 | 2005-04-05 16:01:41+02 | POINT(11.0653331 46.0655397)
          6 | 2005-04-06 00:02:22+02 | POINT(11.0612517 46.0644381)
          6 | 2005-04-06 08:01:18+02 | POINT(11.0656213 46.0667145)
          6 | 2005-04-06 16:03:08+02 |
          6 | 2005-04-07 00:03:08+02 |
          6 | 2005-04-07 08:01:42+02 | POINT(11.0632025 46.0663228)
          6 | 2005-04-07 16:01:41+02 | POINT(11.0643889 46.0661862)
          6 | 2005-04-08 00:01:41+02 | POINT(11.063448 46.0640128)
          6 | 2005-04-08 08:02:21+02 | POINT(11.0659235 46.0660545)
          6 | 2005-04-08 16:03:01+02 | POINT(11.0627981 46.0660227)
          6 | 2005-04-09 00:01:41+02 | POINT(11.0618669 46.0646442)
```

The same with a time interval of 4 h

```
SELECT animals_id, acquisition_time, ST_AsText(geom)
FROM tools.regularize(6, 60*60*4)
LIMIT 15;
```

The first 15 results (out of a total of 191) are

```
 animals_id |    acquisition_time    |          st_astext
------------+------------------------+-----------------------------
          6 | 2005-04-04 08:01:41+02 | POINT(11.0633742 46.0649085)
          6 | 2005-04-04 12:03:04+02 |
          6 | 2005-04-04 16:03:08+02 | POINT(11.0626891 46.0651272)
          6 | 2005-04-04 20:01:17+02 | POINT(11.0645187 46.0646995)
          6 | 2005-04-05 00:03:07+02 |
          6 | 2005-04-05 04:01:03+02 | POINT(11.0622415 46.065877)
          6 | 2005-04-05 08:01:50+02 | POINT(11.063423 46.0648249)
          6 | 2005-04-05 12:03:03+02 | POINT(11.0639178 46.0640381)
          6 | 2005-04-05 16:01:41+02 | POINT(11.0653331 46.0655397)
          6 | 2005-04-05 20:02:48+02 | POINT(11.0634889 46.0651745)
          6 | 2005-04-06 00:02:22+02 | POINT(11.0612517 46.0644381)
          6 | 2005-04-06 04:01:46+02 | POINT(11.0639874 46.0651024)
          6 | 2005-04-06 08:01:18+02 | POINT(11.0656213 46.0667145)
          6 | 2005-04-06 12:01:48+02 | POINT(11.0632134 46.0632785)
          6 | 2005-04-06 16:03:08+02 |
```

And finally, with a time interval of just 1 h

```
SELECT animals_id, acquisition_time, ST_AsText(geom)
FROM tools.regularize(6, 60*60*1)
LIMIT 15;
```

The first 15 results (out of a total of 762) are

```
 animals_id |    acquisition_time    |          st_astext
------------+------------------------+-----------------------------
          6 | 2005-04-04 08:01:41+02 | POINT(11.0633742 46.0649085)
          6 | 2005-04-04 09:01:41+02 |
          6 | 2005-04-04 10:02:24+02 | POINT(11.0626975 46.0637534)
          6 | 2005-04-04 11:01:41+02 |
          6 | 2005-04-04 12:03:04+02 |
          6 | 2005-04-04 13:01:41+02 |
          6 | 2005-04-04 14:00:54+02 | POINT(11.0619604 46.0632978)
          6 | 2005-04-04 15:01:41+02 |
          6 | 2005-04-04 16:03:08+02 | POINT(11.0626891 46.0651272)
          6 | 2005-04-04 17:01:41+02 |
          6 | 2005-04-04 18:03:08+02 | POINT(11.0633284 46.0649574)
          6 | 2005-04-04 19:01:41+02 |
          6 | 2005-04-04 20:01:17+02 | POINT(11.0645187 46.0646995)
          6 | 2005-04-04 21:01:41+02 |
          6 | 2005-04-04 22:02:51+02 | POINT(11.0626356 46.0633533)
```

Interpolation of Missing Coordinates

The next function creates the geometry for the records with no coordinates. It interpolates the positions of the previous and next record, with a weight proportional to the temporal distance. Before you can define the function, you have to create an index that is a sequence number generator[6]. This is used to create temporary table with a name that is always unique in the database:

```
CREATE SEQUENCE tools.unique_id_seq;
COMMENT ON SEQUENCE tools.unique_id_seq
IS 'Sequence used to generate unique numbers for routines that need it (e.g.
functions that need to generate temporary tables with unique names).';
```

You can now create the interpolation function. It accepts as input *animals_id* and a *locations_set* (by default, the *main.view_locations_set*). It checks for all locations with *NULL* geometry to be interpolated. You can also specify a threshold for the allowed time gap between locations with valid coordinates, where the default is two days. If the time gap is smaller, i.e. if you have valid locations before and after the location without coordinates at less than two days of time difference, the new geometry is created, otherwise the *NULL* value is kept (it makes no sense to interpolate if the closest points with valid coordinates are too distant in time).

```
CREATE OR REPLACE FUNCTION tools.interpolate(
  animal integer,  locations_set_name character varying DEFAULT
 'main.view_locations_set'::character varying,
  limit_gap integer DEFAULT 172800)
RETURNS SETOF tools.locations_set AS
$BODY$
DECLARE
  location_set tools.locations_set%rowtype;
  starting_point record;
  ending_point record;
  time_distance_tot integer;
  perc_start double precision;
  x_point double precision;
  y_point double precision;
  var_name character varying;
BEGIN
IF NOT locations_set_name = 'main.view_locations_set' THEN

-- I need a unique name for my temporary table
  SELECT nextval('tools.unique_id_seq')
  INTO var_name;
  EXECUTE
    'CREATE TEMPORARY TABLE
      temp_table_regularize_'|| var_name ||' AS SELECT animals_id,
```

[6] http://www.postgresql.org/docs/9.2/static/sql-createsequence.html.

```
      acquisition_time,
      geom
    FROM
      ' || locations_set_name || '
    WHERE
      animals_id = '|| animal;
  locations_set_name = 'temp_table_regularize_'|| var_name;
END IF;

-- I loop though all the elements of my data set
FOR location_set IN EXECUTE
  'SELECT * FROM ' || locations_set_name || ' WHERE animals_id = ' || animal
LOOP

-- If the record has a NULL geometry values, I look for the previous and
next valid locations and interpolate the coordinates between them
  IF location_set.geom IS NULL THEN
-- I get the geometry and timestamp of the next valid location
    EXECUTE
      'SELECT
        ST_X(geom) AS x_end,
        ST_Y(geom) AS 2y_end,
        extract(epoch FROM acquisition_time) AS ending_time,
        extract(epoch FROM $$' ||location_set.acquisition_time || '$$
        ::timestamp with time zone) AS ref_time
      FROM
        ' || locations_set_name || '
      WHERE
        animals_id = ' || animal || ' AND
        geom IS NOT NULL AND
        acquisition_time > timestamp with time zone $$' ||
        location_set.acquisition_time || '$$
      ORDER BY acquisition_time
      LIMIT 1'
    INTO ending_point;

-- I get the geometry and timestamp of the previous valid location
    EXECUTE
      'SELECT
        ST_X(geom) AS x_start,
        ST_Y(geom) AS y_start,
        extract(epoch FROM acquisition_time) AS starting_time,
        extract(epoch FROM $$' ||location_set.acquisition_time || '$$
        ::timestamp with time zone) AS ref_time
      FROM
        ' || locations_set_name || '
      WHERE
        animals_id = ' || animal || ' AND
        geom IS NOT NULL AND
        acquisition_time < timestamp with time zone $$' ||
        location_set.acquisition_time || '$$
      ORDER BY acquisition_time DESC
      LIMIT 1'
    INTO starting_point;
```

```
-- If both previous and next locations exist, I calculate the interpolated
point, weighting the two points according to the temporal distance to the
location with NULL geometry. The interpolated geometry is calculated
considering lat long as a Cartesian reference. If needed, this approach can
be improved casting geometry as geography and intersecting the line between
previous and next locations with the buffer (from the previous location) at
the given distance.
    IF (starting_point.x_start IS NOT NULL AND ending_point.x_end IS NOT
    NULL) THEN
      time_distance_tot = (ending_point.ending_time -
      starting_point.starting_time);
      IF time_distance_tot <= limit_gap THEN
        perc_start = (starting_point.ref_time -
        starting_point.starting_time)/time_distance_tot;
        x_point = starting_point.x_start + (ending_point.x_end -
        starting_point.x_start) * perc_start;
        y_point = starting_point.y_start + (ending_point.y_end -
        starting_point.y_start) * perc_start;
        SELECT ST_SetSRID(ST_MakePoint(x_point, y_point),4326)
        INTO location_set.geom;
      END IF;
    END IF;
  END IF;
RETURN NEXT location_set;
END LOOP;

-- If I created the temporary table, I delete it here.
IF NOT locations_set_name = 'main.view_locations_set' THEN
  EXECUTE 'drop table ' || locations_set_name;
END IF;
return;
END;
$BODY$
LANGUAGE plpgsql;

COMMENT ON FUNCTION tools.interpolate(integer, character varying, integer)
IS 'This function accepts as input an animals_id and a locations_set (by
default, the main.view_locations_set). It checks for all locations with NULL
geometry. If these locations have previous and next valid locations
(according to the gps_validity_code) with a gap smaller than the defined
threshold (default is 2 days), a new geometry is calculated interpolating
their geometry.';
```

The locations which were interpolated are not marked. You can identify the interpolated locations by joining the result with the original table and see where records originally without coordinates were updated. You can test it comparing the results of the next two queries. In the first one, you just retrieve the original data set:

```
SELECT animals_id, acquisition_time, ST_AsText(geom)
FROM main.view_locations_set
WHERE animals_id = 1 and acquisition_time > '2006-03-01 04:00:00'
```

The first 15 rows of the result (1,486 rows including 398 *NULL* geometries) are

```
 animals_id |    acquisition_time    |          st_astext
------------+------------------------+-----------------------------
          1 | 2006-03-01 05:00:55+01 | POINT(11.0843483 46.010765)
          1 | 2006-03-01 09:02:37+01 | POINT(11.0843323 46.0096131)
          1 | 2006-03-01 13:03:07+01 | POINT(11.0833019 46.0089774)
          1 | 2006-03-01 17:01:55+01 | POINT(11.0831218 46.0090902)
          1 | 2006-03-01 21:02:00+01 | POINT(11.0817527 46.0107692)
          1 | 2006-03-02 01:01:46+01 | POINT(11.0835032 46.0099274)
          1 | 2006-03-02 05:01:12+01 | POINT(11.0830181 46.0101219)
          1 | 2006-03-02 09:01:52+01 | POINT(11.0830582 46.0096292)
          1 | 2006-03-02 13:03:04+01 |
          1 | 2006-03-02 17:01:54+01 | POINT(11.0832821 46.0091515)
          1 | 2006-03-02 21:02:25+01 | POINT(11.0833299 46.0096407)
          1 | 2006-03-03 01:01:18+01 | POINT(11.0847085 46.0105706)
          1 | 2006-03-03 05:01:51+01 | POINT(11.0830901 46.0107184)
          1 | 2006-03-03 09:01:53+01 | POINT(11.0827015 46.0097167)
          1 | 2006-03-03 13:02:40+01 | POINT(11.0831431 46.0088521)
```

In the second query, you can fill the empty geometries using the *tools.interpolate* function:

```
SELECT animals_id, acquisition_time, ST_AsText(geom)
FROM
  tools.interpolate(1,
    '(SELECT *
      FROM main.view_locations_set
      WHERE acquisition_time > ''2006-03-01 04:00:00'')as a')
LIMIT 15;
```

The first 15 rows of the result (same number of records, but *NULL* geometries have been replaced by interpolation) are reported below. You can see that there are no gaps anymore.

```
 animals_id |    acquisition_time    |             st_astext
------------+------------------------+-----------------------------------
          1 | 2006-03-01 05:00:55+01 | POINT(11.0843483 46.010765)
          1 | 2006-03-01 09:02:37+01 | POINT(11.0843323 46.0096131)
          1 | 2006-03-01 13:03:07+01 | POINT(11.0833019 46.0089774)
          1 | 2006-03-01 17:01:55+01 | POINT(11.0831218 46.0090902)
          1 | 2006-03-01 21:02:00+01 | POINT(11.0817527 46.0107692)
          1 | 2006-03-02 01:01:46+01 | POINT(11.0835032 46.0099274)
          1 | 2006-03-02 05:01:12+01 | POINT(11.0830181 46.0101219)
          1 | 2006-03-02 09:01:52+01 | POINT(11.0830582 46.0096292)
          1 | 2006-03-02 13:03:04+01 | POINT(11.0831707019 46.0093891724)
          1 | 2006-03-02 17:01:54+01 | POINT(11.0832821 46.0091515)
          1 | 2006-03-02 21:02:25+01 | POINT(11.0833299 46.0096407)
          1 | 2006-03-03 01:01:18+01 | POINT(11.0847085 46.0105706)
          1 | 2006-03-03 05:01:51+01 | POINT(11.0830901 46.0107184)
          1 | 2006-03-03 09:01:53+01 | POINT(11.0827015 46.0097167)
          1 | 2006-03-03 13:02:40+01 | POINT(11.0831431 46.0088521)
```

You can also use this function in combination with the regularisation function to obtain a regular data set with all valid coordinates. In this query, first you

regularise the function using a time interval of 4 h (for the animal 4), and then, you fill the gap in records with no coordinates:

```
SELECT animals_id, acquisition_time, ST_AsText(geom)
FROM
  tools.interpolate(4,
    '(SELECT *
      FROM tools.regularize(4, 60*60*4)) a')
LIMIT 15;
```

The first 15 records of the result (now 2,854 records with no *NULL* geometries) are

```
 animals_id |    acquisition_time    |             st_astext
------------+------------------------+-----------------------------------
          4 | 2005-10-21 22:00:47+02 | POINT(11.036965 46.0114269)
          4 | 2005-10-22 02:01:24+02 | POINT(11.0359003 46.009527)
          4 | 2005-10-22 06:01:23+02 | POINT(11.0358821 46.0095878)
          4 | 2005-10-22 10:03:07+02 | POINT(11.0363444328 46.0101559523)
          4 | 2005-10-22 14:02:56+02 | POINT(11.0368031 46.0107196)
          4 | 2005-10-22 18:00:43+02 | POINT(11.0358562 46.0093984)
          4 | 2005-10-22 22:01:18+02 | POINT(11.04381 46.0166923)
          4 | 2005-10-23 02:01:41+02 | POINT(11.046664 46.015754)
          4 | 2005-10-23 06:01:24+02 | POINT(11.0467839 46.013193)
          4 | 2005-10-23 10:01:12+02 | POINT(11.0464346 46.0154818)
          4 | 2005-10-23 14:01:42+02 | POINT(11.0467205 46.0155253)
          4 | 2005-10-23 18:00:47+02 | POINT(11.046328920 46.015740574)
          4 | 2005-10-23 22:00:47+02 | POINT(11.0459358396 46.0159566734)
          4 | 2005-10-24 02:00:47+02 | POINT(11.045542758 46.0161727728)
          4 | 2005-10-24 06:00:47+02 | POINT(11.0451496779 46.0163888723)
```

In fact, both functions (as with many other tools for tracking data) have the same information (animal id, acquisition time, geometry) as input and output, so they can be easily nested.

Detection of Sensors Acquisition Scheduling

Another interesting piece of information that can be retrieved from your GPS data set is the sampling frequency scheduling. This information should be available as it is defined by GPS sensors' managers, but in many cases it is not, so it can be useful to derive it from the data set itself. To do so, you have to create a function based on a new data type:

```
CREATE TYPE tools.bursts_report AS (
 animals_id integer,
 starting_time timestamp with time zone,
 ending_time timestamp with time zone,
 num_locations integer,
 num_locations_null integer,
 interval_step integer);
```

This function gives the 'bursts' for a defined animal. Bursts are groups of consecutive locations with the same frequency (or time interval). It requires an animal id and a temporal buffer (in seconds) as input parameters and returns a table with the (supposed) schedule of acquisition frequency. The output table contains the fields *animals_id*, *starting_time*, *ending_time*, *num_locations*, *num_locations_null* and *interval_step* (in seconds, approximated according to multiples of the buffer value). A relocation is considered to have a different interval step if the time gap is greater or less than the defined buffer (the buffer takes into account the fact that small changes can occur because of the delay in reception of the GPS signal). The default value for the buffer is 600 (10 min). The function is directly computed on *main.view_locations_set* (*locations_set* structure) and on the whole data set of the selected animal. Here is the code of the function:

```
CREATE OR REPLACE FUNCTION tools.detect_bursts(
  animal integer,
  buffer integer DEFAULT 600)
RETURNS SETOF tools.bursts_report AS
$BODY$
DECLARE
  location_set tools.locations_set%rowtype;
  cursor_var tools.bursts_report%rowtype;
  starting_time timestamp with time zone;
  ending_time timestamp with time zone;
  location_time timestamp with time zone;
  time_prev timestamp with time zone;
  start_burst timestamp with time zone;
  end_burst timestamp with time zone;
  delta_time integer;
  ref_delta_time integer;
  ref_delta_time_round integer;
  n integer;
  n_null integer;
BEGIN
SELECT min(acquisition_time)
FROM main.view_locations_set
WHERE view_locations_set.animals_id = animal
INTO starting_time;
SELECT max(acquisition_time)
FROM main.view_locations_set
WHERE view_locations_set.animals_id = animal
INTO ending_time;
time_prev = NULL;
ref_delta_time = NULL;
n = 1;
n_null = 0;

FOR location_set IN EXECUTE
  'SELECT animals_id, acquisition_time, geom
  FROM main.view_locations_set
```

```
  WHERE animals_id = '''|| animal ||''' ORDER BY acquisition_time'
LOOP
  location_time = location_set.acquisition_time;
  IF time_prev IS NULL THEN
    time_prev = location_time;
    start_burst = location_time;
  ELSE
    delta_time = (extract(epoch FROM (location_time - time_prev)))::integer;
    IF ref_delta_time IS NULL THEN
      ref_delta_time = delta_time;
      time_prev = location_time;
      end_burst = location_time;
    ELSIF abs(delta_time - ref_delta_time) < (buffer) THEN
      end_burst = location_time;
      time_prev = location_time;
      n = n + 1;
      IF location_set.geom IS NULL then
        n_null = n_null + 1;
      END IF;
    ELSE
      ref_delta_time_round = (ref_delta_time/buffer::double
      precision)::integer * buffer;
      IF ref_delta_time_round = 0 THEN
        ref_delta_time_round = (((extract(epoch FROM (end_burst -
        start_burst)))::integer/n)/60.0)::integer * 60;
      END IF;
    RETURN QUERY SELECT animal, start_burst, end_burst, n, n_null,
    ref_delta_time_round;
    ref_delta_time = delta_time;
    time_prev = location_time;
    start_burst = end_burst;
    end_burst = location_time;
    n = 1;
    n_null = 0;
    END IF;
  END IF;
END LOOP;
ref_delta_time_round = (ref_delta_time/buffer::double precision)::integer *
buffer;
IF ref_delta_time_round = 0 THEN
  ref_delta_time_round = ((extract(epoch FROM end_burst - start_burst))::
  integer/n)::integer;
END IF;
RETURN QUERY SELECT animal, start_burst, end_burst, n , n_null,
  ref_delta_time_round;
RETURN;
END;
$BODY$
LANGUAGE plpgsql;
```

```
COMMENT ON FUNCTION tools.detect_bursts(integer, integer)
IS 'This function gives the "bursts" for a defined animal. Bursts are groups
of consecutive locations with the same frequency (or time interval). It
receives an animal id and a buffer (in seconds) as input parameters and
returns a table with the (supposed) schedule of location frequencies. The
output table has the fields: animals_id, starting_time, ending_time,
num_locations, num_locations_null, and interval_step (in seconds,
approximated according to multiples of the buffer value). A relocation is
considered to have a different interval step if the time gap is greater or
less than the defined buffer (the buffer takes into account the fact that
small changes can occur because of the delay in receiving the GPS signal).
The default value for the buffer is 600 (10 minutes). The function is
directly computed on main.view_locations_set (locations_set structure) and
on the whole data set for the selected animal.';
```

Here, you can verify the results. You can use the function with animal 5:

```
SELECT
  animals_id AS id,
  starting_time,
  ending_time,
  num_locations AS num,
  num_locations_null AS num_null,
  (interval_step/60.0/60)::numeric(5,2) AS hours
FROM
  tools.detect_bursts(5);
```

The result is

```
 id |     starting_time      |      ending_time       | num  | nulls | hours
----+------------------------+------------------------+------+-------+-------
  5 | 2006-11-12 13:03:04+01 | 2007-10-28 05:01:17+01 | 2098 |   193 |  4.00
  5 | 2007-10-28 05:01:17+01 | 2007-10-29 13:01:23+01 |    1 |     0 | 32.00
  5 | 2007-10-29 13:01:23+01 | 2008-03-07 05:00:49+01 |  778 |    29 |  4.00
  5 | 2008-03-07 05:00:49+01 | 2008-03-07 21:03:07+01 |    1 |     0 | 16.00
  5 | 2008-03-07 21:03:07+01 | 2008-03-15 09:01:37+01 |   45 |     5 |  4.00
```

In this case, the time interval is constant (14,400 s, which means 4 h). The second and fourth bursts are made of a single location. This is because you have a gap greater than the temporal buffer with no records, not a real new burst.

Now run the same function on animal 6:

```
SELECT
  animals_id AS id,
  starting_time,
  ending_time,
  num_locations AS num,
  num_locations_null AS num_null,
  (interval_step/60.0/60)::numeric(5,2) AS hours
FROM
  tools.detect_bursts(6);
```

The result is reported below. In this case, a more varied scheduling has been used (1, 2 and 4 h):

```
 id |     starting_time      |      ending_time       | num | nulls | hours
----+------------------------+------------------------+-----+-------+-------
  6 | 2005-04-04 08:01:41+02 | 2005-04-13 06:00:48+02 | 107 |    16 |  2.00
  6 | 2005-04-13 06:00:48+02 | 2005-04-13 10:02:24+02 |   1 |     0 |  4.00
  6 | 2005-04-13 10:02:24+02 | 2005-04-14 02:02:18+02 |   8 |     0 |  2.00
  6 | 2005-04-14 02:02:18+02 | 2005-04-29 02:00:54+02 |  90 |     3 |  4.00
  6 | 2005-04-29 02:00:54+02 | 2005-05-04 22:01:23+02 |  70 |     1 |  2.00
  6 | 2005-05-04 22:01:23+02 | 2005-05-05 03:01:46+02 |   1 |     0 |  5.00
  6 | 2005-05-05 03:01:46+02 | 2005-05-06 01:01:47+02 |  22 |     2 |  1.00
```

Representations of Home Ranges

Home range is another representation of animal movement and behaviour that can be derived from GPS tracking data. Home range is roughly described as the area in which an animal normally lives and travels, excluding migration, emigration or other large infrequent excursions. There are different ways to define this concept and different methods for computing it. A common approach to modelling home ranges is the delineation of the boundaries (polygons) of the area identified (according to a specific definition) as home range. The simplest way to create a home range is the MCP approach. PostGIS has a specific function to compute MCP (*ST_ConvexHull*). In this example, you can create a function to produce an MCP using just a percentage of the available locations, in order to exclude the outliers which are far from the pool of locations, based on a starting and ending acquisition time. First, you can create a table where data can be stored. This table also includes some additional information that describes the result and can be used both to document it and to run meta-analysis. In this way, all the results of your analysis are permanently stored, accessible, compact and documented.

```
CREATE TABLE analysis.home_ranges_mcp (
  home_ranges_mcp_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,
  description character varying,
  ref_user character varying,
  num_locations integer,
  area numeric(13,5),
  geom geometry (multipolygon, 4326),
  percentage double precision,
  insert_timestamp timestamp with time zone DEFAULT timezone('UTC'::text,
  ('now'::text)::timestamp(0) with time zone),
  original_data_set character varying,
  CONSTRAINT home_ranges_mcp_pk
    PRIMARY KEY (home_ranges_mcp_id),
```

```
  CONSTRAINT home_ranges_mcp_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_mcp
IS 'Table that stores the home range polygons derived from MCP. The area is
computed in hectars.';

CREATE INDEX fki_home_ranges_mcp_animals_fk
  ON analysis.home_ranges_mcp
  USING btree (animals_id);
CREATE INDEX gist_home_mcp_ranges_index
  ON analysis.home_ranges_mcp
  USING gist (geom);
```

This function applies the MCP algorithm (also called convex hull) to a set of locations. The input parameters are the animal id (each analysis is related to a single individual), the percentage of locations to be considered and a *locations_set* object (the default is *main.view_locations_set*). An additional parameter can be added: a description that will be included in the table *home_ranges_mcp*, where the result of the analysis is stored. The parameter *percentage* defines how many locations are included in the analysis: if, for example, 90 % is specified (as 0.9), the 10 % of locations farthest from the centroid of the data set will be excluded. If no parameters are specified, the percentage of 100 % is used and the complete data set (from the first to the last location) are considered. The following creates the function:

```
CREATE OR REPLACE FUNCTION tools.mcp_perc(
  animal integer,
  perc double precision DEFAULT 1,
  description character varying DEFAULT 'Standard analysis'::character
  varying,locations_set_name character varying DEFAULT
  'main.view_locations_set'::character varying,starting_time
   timestamp with time zone DEFAULT NULL::timestamp with time zone,
   ending_time timestamp with time zone  DEFAULT NULL::timestamp with time
  zone)
RETURNS integer AS
$BODY$
DECLARE
  hr record;
  var_name character varying;
  locations_set_name_input character varying;
BEGIN
  locations_set_name_input = locations_set_name;
```

```
IF NOT locations_set_name = 'main.view_locations_set' THEN
  SELECT nextval('tools.unique_id_seq') INTO var_name;
  EXECUTE
    'CREATE TEMPORARY TABLE temp_table_mcp_perc_'|| var_name ||' AS
      SELECT *
      FROM ' || locations_set_name || '
      WHERE animals_id = '|| animal;
      locations_set_name = 'temp_table_mcp_perc_'|| var_name;
END IF;

IF perc <= 0 OR perc > 1 THEN
  RAISE EXCEPTION 'INVALID PARAMETER: the percentage of the selected
  (closest to the data set centroid) points must be a value > 0 and <= 1';
END IF;

IF starting_time IS NULL THEN
  EXECUTE
    'SELECT min(acquisition_time)
    FROM '|| locations_set_name ||'
    WHERE '|| locations_set_name ||'.animals_id = ' || animal || ' AND '||
    locations_set_name ||'.geom IS NOT NULL '
    INTO starting_time;
END IF;

IF ending_time IS NULL THEN
  EXECUTE
    'SELECT max(acquisition_time)
    FROM '|| locations_set_name ||'
    WHERE '|| locations_set_name ||'.animals_id = ' || animal || ' AND '||
    locations_set_name ||'.geom IS NOT NULL '
    INTO ending_time;
END IF;

EXECUTE
  'SELECT
    animals_id,
    min(acquisition_time) AS start_time,
    max(acquisition_time) AS end_time,
    count(animals_id) AS num_locations,
    ST_Area(geography(ST_ConvexHull(ST_Collect(a.geom)))) AS area,
    (ST_ConvexHull(ST_Collect(a.geom))).ST_Multi AS geom
  FROM
    (SELECT '|| locations_set_name ||'.animals_id, '|| locations_set_name
    ||'.geom, acquisition_time, ST_Distance('|| locations_set_name ||'.geom,
      (SELECT ST_Centroid(ST_collect('|| locations_set_name ||'.geom))
       FROM '|| locations_set_name ||'
       WHERE '|| locations_set_name ||'.animals_id = ' || animal || ' AND '||
       locations_set_name ||'.geom IS NOT NULL AND '|| locations_set_name ||
       '.acquisition_time >= $$' || starting_time ||'$$::timestamp with time
       zone AND '|| locations_set_name ||'.acquisition_time <= $$' ||
       ending_time || '$ $::timestamp with time zone
       GROUP BY '|| locations_set_name ||'.animals_id)) AS dist
```

```
    FROM '|| locations_set_name ||'
    WHERE '|| locations_set_name ||'.animals_id = ' || animal || ' AND '||
    locations_set_name ||'.geom IS NOT NULL AND '|| locations_set_name
    ||'.acquisition_time >= $$' || starting_time ||'$$::timestamp with time
    zone and '|| locations_set_name ||'.acquisition_time <= $$' ||
    ending_time || '$ $::timestamp with time zone
    ORDER BY
     ST_Distance('|| locations_set_name ||'.geom,
    (SELECT ST_Centroid(ST_Collect('|| locations_set_name ||'.geom))
     FROM '|| locations_set_name ||'
     WHERE '|| locations_set_name ||'.animals_id = ' || animal || ' AND '||
     locations_set_name ||'.geom IS NOT NULL AND '|| locations_set_name
     ||'.acquisition_time >= $$' || starting_time ||'$$::timestamp with time
     zone and '|| locations_set_name ||'.acquisition_time <= $$' ||
     ending_time || '$ $::timestamp with time zone
     GROUP BY '|| locations_set_name ||'.animals_id))LIMIT ((
    (SELECT count('|| locations_set_name ||'.animals_id) AS count
     FROM '|| locations_set_name ||'
     WHERE '|| locations_set_name ||'.animals_id = ' || animal || ' AND '||
     locations_set_name ||'.geom IS NOT NULL AND '|| locations_set_name
     ||'.acquisition_time >= $$' || starting_time ||'$$::timestamp with time
     zone AND '|| locations_set_name ||'.acquisition_time <= $$' ||
     ending_time  || '$ $::timestamp with time zone ))::numeric * '
     || perc || ')::integer) a
  GROUP BY a.animals_id;'
  INTO hr;
  IF hr.num_locations < 3 or hr.num_locations IS NULL THEN
    RAISE NOTICE 'INVALID SELECTION: less then 3 points or no points at all
    match the given criteria. The animal % will be skipped.', animal;
RETURN 0;
END IF;
INSERT INTO analysis.home_ranges_mcp (animals_id, start_time, end_time,
  percentage, description, ref_user, num_locations,area, geom,
  original_data_set)values (animal, starting_time, ending_time , perc ,
  description,current_user, hr.num_locations, hr.area/1000000.00000, hr.geom,
  locations_set_name_input);
  IF NOT locations_set_name = 'main.view_locations_set' THEN
  EXECUTE 'drop table ' || locations_set_name;
  END IF;
  RAISE NOTICE 'Operation correctly performed. Record inserted into
    analysis.home_ranges % ', animal;
  RETURN 1;
  END;
  $BODY$
  LANGUAGE plpgsql;
  COMMENT ON FUNCTION tools.mcp_perc(integer, double precision, character
  varying, character varying, timestamp with time zone, timestamp with time
  zone)
  IS 'This function applies the MCP (Minimum Convex Polygon) algorithm (also
  called convex hull) to a set of locations. The input parameters are the
```

```
animal id (each analysis is related to a single individual), the percentage
of locations considered, a locations_set object (the default is
main.view_locations_set). An additional parameter can be added: a description
that will be included in the table home_ranges_mcp, where the result of  the
analysis is stored. The parameter "percentage" defines how many locations are
included in the analysis: if for example 90% is specified (as 0.9),  the 10%
of locations farthest from the centroid of the data set will be excluded.If
no parameters are specified, percentage of 100% is used and the complete data
set (from the first to the last location) are considered. The function, once
computed the MCP and stored the result in home_range_mcp, does not return any
thing. A few constraints to prevent errors are included (no points selected,
percentage out of range). Note that this function works with a fixed centroid,
computed at the beginning, so the distance is calculated on this basis   for
the entire selection process.';
```

You can create the MCP at different percentage levels:

```
SELECT tools.mcp_perc(1, 0.1, 'test 0.1');
SELECT tools.mcp_perc(1, 0.5, 'test 0.5');
SELECT tools.mcp_perc(1, 0.75, 'test 0.75');
SELECT tools.mcp_perc(1, 1, 'test 1');
SELECT tools.mcp_perc(1, 1, 'test start and end', 'main.view_locations_set',
  '2006-01-01 00:00:00', '2006-01-10 00:00:00');
SELECT tools.mcp_perc(animals_id, 0.9, 'test all animals at 0.9') FROM
  main.animals;
```

The output is stored in the table. You can retrieve part of the columns of the table with

```
SELECT
  home_ranges_mcp_id AS id, animals_id AS animal, description, num_locations
  AS num, area, percentage
FROM
  analysis.home_ranges_mcp;
```

The result is

```
 id | animal |       description       | num  |  area   | percentage
----+--------+-------------------------+------+---------+------------
  1 |      1 | test 0.1                |  165 | 0.91037 |        0.1
  2 |      1 | test 0.5                |  824 | 3.12442 |        0.5
  3 |      1 | test 0.75               | 1235 | 4.52416 |       0.75
  4 |      1 | test 1                  | 1647 | 8.08596 |          1
  5 |      1 | test start and end      |   37 | 0.18170 |          1
  6 |      1 | test all animals at 0.9 | 1482 | 5.25487 |        0.9
  7 |      2 | test all animals at 0.9 | 1975 | 9.03271 |        0.9
  8 |      3 | test all animals at 0.9 | 1643 | 8.93319 |        0.9
  9 |      4 | test all animals at 0.9 | 2377 | 9.74893 |        0.9
 10 |      5 | test all animals at 0.9 | 2426 | 6.57880 |        0.9
 11 |      6 | test all animals at 0.9 |  250 | 0.13362 |        0.9
```

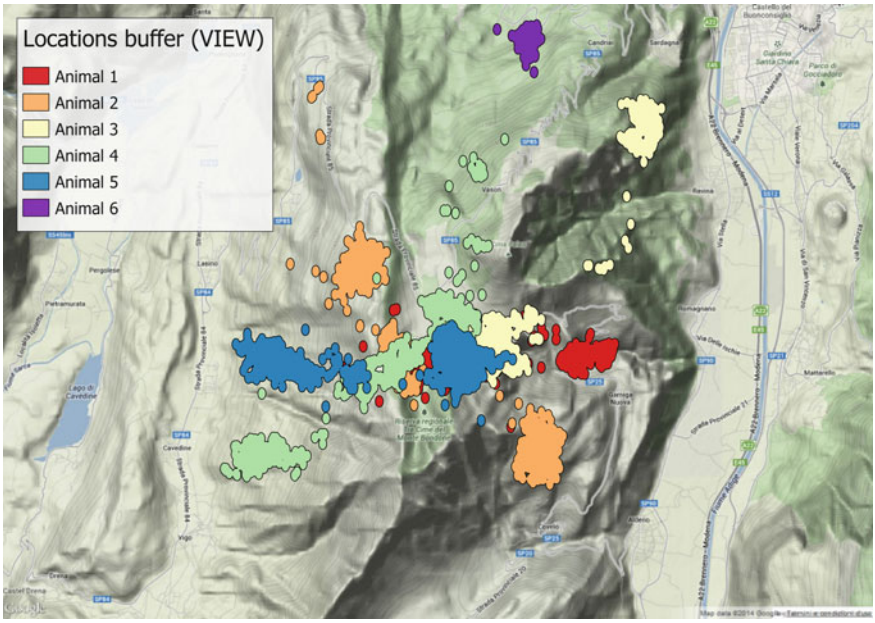

Fig. 9.1 Layer with dissolved buffers around GPS locations

Note that the last statement generates the MCP for all the animals with a single command.

A further example of synthetic representation of the GPS location set is illustrated in the view below: for each GPS position, you can compute a buffer (a circle of 0.001 degrees, which at this latitude corresponds to about 100 meters), and then, all the buffers of the same animal are merged together:

```
CREATE VIEW analysis.view_locations_buffer AS
  SELECT
    animals_id,
    ST_Union(ST_Buffer(geom, 0.001))::geometry(multipolygon, 4326) AS geom
  FROM main.gps_data_animals
  WHERE gps_validity_code = 1
  GROUP BY animals_id
  ORDER BY animals_id;
COMMENT ON VIEW analysis.view_locations_buffer
IS 'GPS locations - Buffer (dissolved) of 0.001 degrees.';
```

As you can see, when you visualise it (Fig. 9.1), the view, which is a query run every time you access the view, takes some time, as quite complex operations must be performed. If used often, it could be transformed into a permanent table (with *CREATE TABLE* command). In this case, you might also want to add keys and indexes.

Geometric Parameters of Animal Movements

Another type of analytical tool that can be implemented within the database is the computation of the geometric parameters of trajectories (e.g. spatial and temporal distance between locations, speed and angles). As the meaning of these parameters changes with the time step, you will create a function that computes the parameters just for steps that have a time gap equal to a value defined by the user. First, you must create the new data type *tools.geom_parameters*:

```
CREATE TYPE tools.geom_parameters AS(
  animals_id integer,
  acquisition_time timestamp with time zone,
  acquisition_time_t_1 timestamp with time zone,
  acquisition_time_t_2 timestamp with time zone,
  deltat_t_1 integer,
  deltat_t_2 integer,
  deltat_start integer,
  dist_t_1 integer,
  dist_start integer,
  speed_mh_t_1 numeric(8,2),
  abs_angle_t_1 numeric(7,5),
  rel_angle_t_2 numeric(7,5));
```

Now you can create the function *tools.geom_parameters*. It returns a table with the geometric parameters of the data set (reference: previous location): time gap with the previous point, time gap with the previous–previous point, distance to the previous point, speed of the last step, distance to the first point of the data set, absolute angle (from the previous location), relative angle (from the previous and previous–previous locations). The input parameters are the animal id, the time gap and a buffer to take into account possible time differences due to GPS data reception. The time gap parameter selects just locations that have the previous point at the defined time interval (with a buffer tolerance). All the other locations are not taken into consideration. A *locations_set* class is accepted as the input table. It is also possible to specify the starting and ending acquisition time of the time series. The output is a table with the structure *geom_parameters*. If you want to calculate the geometric parameters of an irregular sequence (i.e. the parameters calculated in relation to the previous/next location regardless of the regularity of the time gap), you can use a plain SQL based on window functions[7] with no need for customised functions. It is important to note that while a step is the movement between two points, in many cases the geometric parameters of the movement (step) are associated with the starting or the ending point. In this book, we use the

[7] http://www.postgresql.org/docs/9.2/static/tutorial-window.html.

ending point as reference. In some software, particularly the adehabitat[8] package for R (see Chap. 10), the step is associated with the starting point. If needed, the queries and functions presented here can be modified to follow this convention. The code of the function is

```
CREATE OR REPLACE FUNCTION tools.geom_parameters(
  animal integer,
  time_interval integer DEFAULT 10800,
  buffer double precision DEFAULT 600,
  locations_set_name character varying DEFAULT
  'main.view_locations_set'::character varying,
  starting_time timestamp with time zone DEFAULT NULL::timestamp with time zone,
  ending_time timestamp with time zone DEFAULT NULL::timestamp with time zone)
RETURNS SETOF tools.geom_parameters AS
$BODY$
DECLARE
  cursor_var tools.geom_parameters%rowtype;
  check_animal boolean;
  var_name character varying;
BEGIN
EXECUTE
  'SELECT ' || animal || ' IN
    (SELECT animals_id FROM main.animals)' INTO check_animal;

IF NOT check_animal THEN
  RAISE EXCEPTION 'This animal is not in the data set...';
END IF;

IF starting_time IS NULL THEN
  SELECT min(acquisition_time)
  FROM main.view_locations_set
  WHERE view_locations_set.animals_id = animal
  INTO starting_time;
END IF;

IF ending_time IS NULL THEN
  SELECT max(acquisition_time)
  FROM main.view_locations_set
  WHERE view_locations_set.animals_id = animal
  INTO ending_time;
END IF;

IF NOT locations_set_name = 'main.view_locations_set' THEN
  SELECT nextval('tools.unique_id_seq') into var_name;
  EXECUTE
    'CREATE TEMPORARY TABLE temp_table_temp_table_geoparameters_'|| var_name
    ||' AS
      SELECT animals_id, acquisition_time, geom
      FROM ' || locations_set_name || '
      WHERE animals_id = '|| animal;
      locations_set_name = 'temp_table_temp_table_geoparameters_'|| var_name;
END IF;
```

[8] http://cran.r-project.org/web/packages/adehabitat/index.html.

```
FOR cursor_var IN EXECUTE
  'SELECT
    animals_id,
    acquisition_time,
    acquisition_time_t_1,
    acquisition_time_t_2,
    deltaT_t_1,
    deltaT_t_2,
    deltaT_start,
    dist_t_1,
    dist_start,
    speed_Mh_t_1,
    abs_angle_t_1,
    CASE WHEN (deltaT_t_2 < ' || time_interval * 2 + buffer || ' and
      deltaT_t_2 > ' || time_interval * 2 - buffer || ') THEN
      rel_angle_t_2
    ELSE
      NULL
    END
  FROM
    (SELECT
      animals_id,
      acquisition_time,
      lead(acquisition_time,-1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time) AS acquisition_time_t_1,
      lead(acquisition_time,-2) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time) AS acquisition_time_t_2,
      rank() OVER (PARTITION BY animals_id ORDER BY acquisition_time),
      (extract(epoch FROM acquisition_time) - lead(extract(epoch FROM
      acquisition_time), -1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))::integer AS deltat_t_1,
      (extract(epoch FROM acquisition_time) - lead(extract(epoch FROM
      acquisition_time), -2) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))::integer AS deltat_t_2,
      (extract(epoch FROM acquisition_time) - first_value(extract(epoch FROM
      acquisition_time)) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))::integer AS deltat_start,
      (ST_Distance_Spheroid(geom, lead(geom, -1) OVER (PARTITION BY
      animals_id ORDER BY acquisition_time), ''SPHEROID["WGS
      84",6378137,298.257223563]''))::integer AS dist_t_1,
      ST_Distance_Spheroid(geom, first_value(geom) OVER (PARTITION BY
      animals_id ORDER BY acquisition_time), ''SPHEROID["WGS
      84",6378137,298.257223563]'')::integer AS dist_start,
      (ST_Distance_Spheroid(geom, lead(geom, -1) OVER (PARTITION BY
      animals_id ORDER BY acquisition_time), ''SPHEROID["WGS
      84",6378137,298.257223563]'')/(extract(epoch FROM acquisition_time) -
      lead(extract(epoch FROM acquisition_time), -1) OVER (PARTITION BY
      animals_id ORDER BY acquisition_time))*60*60)::numeric(8,2) AS
      speed_Mh_t_1, ST_Azimuth(geom::geography, (lead(geom, -1) OVER
      (PARTITION BY animals_id ORDER BY acquisition_time))::geography) AS
      abs_angle_t_1, ST_Azimuth(geom::geography, (lead(geom, -1) OVER
      (PARTITION BY  animals_id ORDER BY acquisition_time))::geography) -
      ST_Azimuth((lead(geom, -1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))::geography, (lead(geom, -2) OVER (PARTITION BY
      animals_id ORDER BY acquisition_time))::geography) AS rel_angle_t_2
    FROM
```

```
    FROM
      '|| locations_set_name ||'
    WHERE
      animals_id = ' || animal ||' AND
      geom IS NOT NULL AND
      acquisition_time >= ''' || starting_time || ''' AND
      acquisition_time <= ''' || ending_time || ''') a
  WHERE
    deltaT_t_1 <' || time_interval + buffer || ' AND
    deltaT_t_1 > '|| time_interval - buffer
LOOP
RETURN NEXT cursor_var;
END LOOP;

IF NOT locations_set_name = 'main.view_locations_set' THEN
  EXECUTE 'drop table ' || locations_set_name;
END IF;
RETURN;
END;
$BODY$
LANGUAGE plpgsql;

COMMENT ON FUNCTION tools.geom_parameters(integer, integer, double
precision, character varying, timestamp with time zone, timestamp with time
zone)
IS 'This function returns a table with the geometric parameters of the data
set (reference: previous location): time gap with the previous point, time
gap with the previous-previous point, distance to the previous point, speed
of the last step, distance from the first point of the data set, absolute
angle (from the previous location), relative angle (from the previous and
previous-previous locations). The input parameters are the animal id, the
time gap, and the buffer. The time gap selects just locations that have the
previous point at a defined time interval (with a buffer tolerance). All the
other points are not taken into consideration. A locations_set class is
accepted as the input table. It is also possible to specify the starting and
ending acquisition time of the time series. The output is a table with the
structure geom_parameters.';
```

To test how the function works, you can run and compare the function applied to the same animal 6 at different time steps. In the first case, you can use 2 h:

```
SELECT * FROM tools.geom_parameters(6, 60 * 60 * 2, 600);
```

A subset of the columns of the first 10 rows returned by the function is

acqtime	acqtime_1	acqtime_2	d_1	d_start	abs_ang	rel_ang
94 10:02:24	94 08:01:41		139	139	0.39	
94 16:03:08	94 14:00:54	94 10:02:24	211	58	3.41	
94 18:03:08	94 16:03:08	94 14:00:54	53	6	5.08	1.66
94 20:01:17	94 18:03:08	94 16:03:08	96	92	5.01	-0.06
94 22:02:51	94 20:01:17	94 18:03:08	209	182	0.77	-4.24
95 04:01:03	95 02:01:40	94 22:02:51	179	139	3.19	
95 06:02:15	95 04:01:03	95 02:01:40	218	171	4.70	1.52
95 08:01:50	95 06:02:15	95 04:01:03	174	10	0.82	-3.89
95 10:01:49	95 08:01:50	95 06:02:15	266	272	0.47	-0.34
95 12:03:03	95 10:01:49	95 08:01:50	218	105	3.96	3.49

The real results include a longer list of parameters that is not possible to report because of space constraints. To save space, the dates have been transformed into Julian day of the year (DOY, in the range 1–365).

You can apply the function with an interval step of 4 h:

```
SELECT * FROM tools.geom_parameters(6, 60 * 60 * 4, 600);
```

A subset of the result is reported below:

acqtime	acqtime_1	acqtime_2	d_1	d_start	abs_ang	rel_ang
94 14:00:54	94 10:02:24	94 08:01:41	76	210	0.84	
95 02:01:40	94 22:02:51	94 20:01:17	109	119	2.78	
96 18:01:50	96 14:01:24	96 12:01:48	216	44	3.37	
97 02:02:08	96 22:01:48	96 20:02:20	233	302	0.11	
97 12:01:55	97 08:01:42	97 06:00:54	327	179	0.17	
97 20:01:00	97 16:01:41	97 14:01:52	182	20	0.48	
98 12:02:56	98 08:02:21	98 02:01:54	338	108	0.88	
99 04:02:13	99 00:01:41	98 22:01:22	87	83	3.70	
99 12:03:07	99 08:03:06	99 06:02:17	87	288	1.76	
99 16:00:54	99 12:03:07	99 08:03:06	428	146	3.29	1.54

As you can see, there are very few sequences of at least three points at a regular temporal distance of 4 h in the original data set (at least in the first records).

Now apply the function with 8 h interval step:

```
SELECT * FROM tools.geom_parameters(6, 60*60*8, 600);
```

The result is reported below. Just 3 records are retrieved because the scheduling of 8 h is not used in this data set.

acqtime	acqtime_1	acqtime_2	d_1	d_start	abs_ang	rel_ang
108 18:01:45	108 10:02:18	108 06:02:52	53	114	0.09	
112 22:01:59	112 14:03:04	112 10:01:46	252	121	3.42	
117 10:01:01	117 02:03:05	116 22:00:53	181	53	2.88	

An Alternative Representation of Home Ranges

In the next example of possible methods to represent and analyse GPS locations using the tools provided by PostgreSQL and PostGIS, you can create a grid surface and calculate an estimation of the time spent in seconds by each animal within each 'pixel'. There are many existing approaches to producing this information; in this case, you will use an algorithm that is conceptually similar to a simplified Brownian bridge method (Horne et al. 2007) and to the method proposed in (Kranstauber et al. 2012). In this example, you can assume that the animal moves with along the trajectory described by the temporal sequence of locations and that the speed is constant along each step. You can create a grid with the given resolution that is intersected with the trajectory. For each segment of the trajectory generated by the intersection, the time spent by the animal is calculated (considering the time interval of that step and the relative length of the segment compared to the whole step length). Finally, you can sum the time spent in all the segments inside each cell. You can implement this method using a view and a function that creates the grid, which is based on a new data type that you create with the code

```
CREATE TYPE tools.grid_element AS (cell_id integer, geom geometry);
```

Then, you can create the grid function:

```
CREATE OR REPLACE FUNCTION tools.create_grid(
  locations_collection geometry, xysize integer)
RETURNS SETOF tools.grid_element AS
$BODY$

WITH spatial_object AS
  (SELECT
    ST_Xmin(ST_Transform($1,tools.srid_utm(ST_X(ST_Centroid($1)),
    ST_Y(ST_Centroid($1))))::integer AS xmin,
    ST_Ymin(ST_Transform($1,tools.srid_utm(ST_X(ST_Centroid($1)),
    ST_Y(ST_Centroid($1))))::integer AS ymin,
    ST_Xmax(ST_Transform($1,tools.srid_utm(ST_X(ST_Centroid($1)),
    ST_Y(ST_Centroid($1))))::integer AS xmax,
    ST_Ymax(ST_Transform($1,tools.srid_utm(ST_X(ST_Centroid($1)),
    ST_Y(ST_Centroid($1))))::integer AS ymax,
    tools.srid_utm(ST_X(ST_Centroid($1)), ST_Y(ST_Centroid($1))) AS sridset)
  SELECT
    (ROW_NUMBER() OVER ())::integer,
    ST_Translate(cell, i , j)
  FROM
    generate_series(
      ((((SELECT xmin FROM spatial_object) - $2/2)/100)::integer)*100,
      (SELECT xmax FROM spatial_object) + $2, $2) AS i,
    generate_series(
```

```
      ((((SELECT ymin FROM spatial_object) - $2/2)/100)::integer)*100,
      (SELECT ymax FROM spatial_object) + $2, $2) AS j, spatial_object,
      (SELECT ST_setsrid(ST_GeomFROMText('POLYGON((0 0, 0 '||$2||', '||$2||'
      '|| $2||', '||$2||' 0,0 0))'),
      (SELECT sridset FROM spatial_object)) AS cell) AS foo;
$BODY$
LANGUAGE sql;

COMMENT ON FUNCTION tools.create_grid(geometry, integer)
IS 'Function that creates a vector grid with a given resolution that
contains a given geometry.';
```

Now, you can create the view that generates the probability surface (in this example, for the animal 1 with a grid with a resolution of 100 m):

```
CREATE OR REPLACE VIEW analysis.view_probability_grid_traj AS
  WITH
  setx AS (
    SELECT
      gps_data_animals.gps_data_animals_id,
      gps_data_animals.animals_id,
      ST_MakeLine(gps_data_animals.geom,
      lead(gps_data_animals.geom, (-1)) OVER (PARTITION BY
      gps_data_animals.animals_id ORDER BY gps_data_animals.acquisition_time))
      AS geom, ST_Length(ST_MakeLine(gps_data_animals.geom,
      lead(gps_data_animals.geom, (-1)) OVER (PARTITION BY
      gps_data_animals.animals_id ORDER BY
      gps_data_animals.acquisition_time))::geography) AS line_length,
      CASE WHEN (date_part('epoch'::text, gps_data_animals.acquisition_time)
      - date_part('epoch'::text, lead(gps_data_animals.acquisition_time, (-1))
      OVER (PARTITION BY gps_data_animals.animals_id ORDER BY
      gps_data_animals.acquisition_time))) < (60 * 60 * 24)::double precision
      THEN date_part('epoch'::text, gps_data_animals.acquisition_time) -
      date_part('epoch'::text, lead(gps_data_animals.acquisition_time, (-1))
      OVER (PARTITION BY gps_data_animals.animals_id ORDER BY
      gps_data_animals.acquisition_time))
      ELSE
        0::double precision
      END AS time_spent
    FROM
      main.gps_data_animals
    WHERE
      gps_data_animals.gps_validity_code = 1 AND
      (gps_data_animals.animals_id = 1)
    ORDER BY
      gps_data_animals.acquisition_time),

  gridx AS (
    SELECT
```

```
    setx.animals_id,
    tools.create_grid(ST_Collect(setx.geom), 100) AS cell
  FROM setx
  GROUP BY setx.animals_id)
  SELECT
    a.animals_id * 10000 + a.cell_id AS id,
    a.animals_id,
    a.cell_id,
    ST_Transform(a.geom, 4326)::geometry(Polygon,4326) AS geom,
    (sum(a.segment_time_spent) / 60::double precision / 60::double
    precision)::integer AS hours_spent
  FROM
    (SELECT
      gridx.animals_id,
      (gridx.cell).cell_id AS cell_id,
      CASE setx.line_length WHEN 0 THEN
        setx.time_spent
      ELSE
        setx.time_spent * ST_Length(ST_Intersection(ST_Transform(setx.geom,
        ST_SRID((SELECT (gridx.cell).geom AS geom FROM gridx LIMIT 1))),
        (gridx.cell).geom)) / setx.line_length
      END AS segment_time_spent,
      (gridx.cell).geom AS geom
    FROM gridx, setx
    WHERE ST_Intersects(ST_Transform(setx.geom, ST_SRID((SELECT
    (gridx.cell).geom AS geom FROM gridx LIMIT 1))), (gridx.cell).geom) AND
    setx.time_spent > 0::double precision AND setx.animals_id =
    gridx.animals_id) a
    GROUP BY a.animals_id, a.cell_id, a.geom
    HAVING sum(a.segment_time_spent) > 0::double precision;

COMMENT ON VIEW analysis.view_probability_grid_traj
IS 'This view presents the SQL code to calculate the time spent by an animal
on every cell of a grid with a defined resolution, which corresponds to a
probability surface. Trajectories (segments between locations) are considered.
Each segment represents the time spent between the two locations. This view
calls the function tools.reate_grid. This is a view with pure SQL, but this
tool can be coded into a function that uses temporary tables and some other
optimized approaches in order to speed up the processing time. In this case,
just animal 1 is returned.';
```

This process involves time-consuming computation and you might need to wait several seconds to get the result (Fig. 9.2).

This approach has a number of advantages:

- it is implemented with SQL, which is a relatively simple language to modify/customise/extend;
- it is run inside the database, so results can be directly stored in a table, used to run meta-analysis, and extended using other database tools;
- it is conceptually simple and gives a 'real' measure (time spent in terms of hours);
- no parameters with unclear physical meaning have to be set; and
- it handles heterogeneous time intervals.

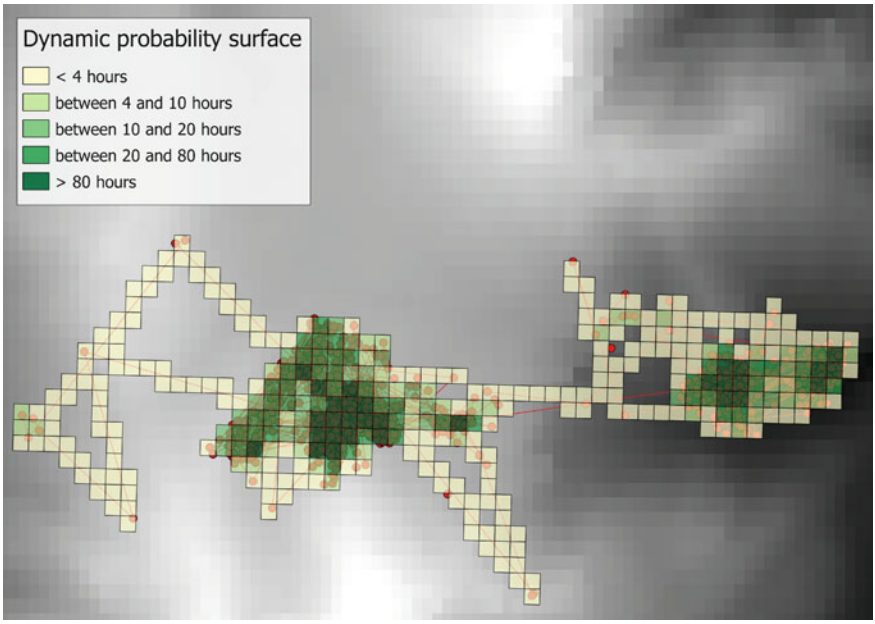

Fig. 9.2 Probability surface with *analysis.view_probability_grid_traj* (a DEM is visualised in the background)

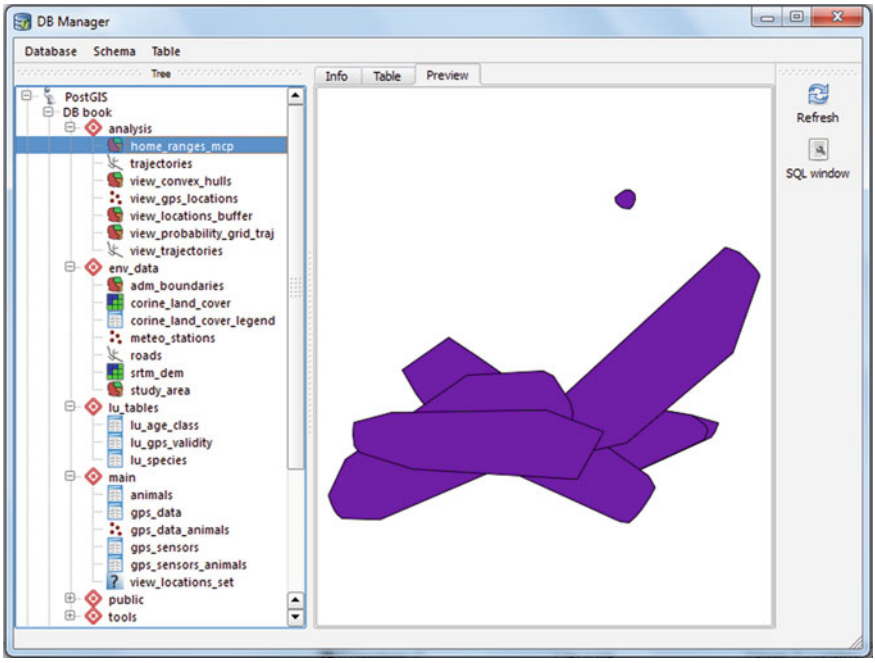

Fig. 9.3 Spatial content of the database as seen from DB Manager in QGIS

On the other hand, it implicitly relies on a very simplified movement model (the animal moves along the segment that connects two locations with a constant speed).

Figure 9.3 shows a picture of the spatial content of the database in QGIS (DB Manager).

Dynamic Age Class

While age class is stored in the *animals* table with reference to the capture time, it can change over time. If this information must be associated with each location (according to the acquisition time), a dynamic calculation of the age class must be used. We present here an example valid for roe deer. With a conservative approach, we can consider that on 1 April of each year, all the animals that were fawns become yearlings, and all the yearlings become adults. Adults remain adults. The function below requires an animal id and an acquisition time as input. Then, it checks the capture date and the age class at capture. Finally, it compares the capture time to the acquisition time: if 1 April has been 'crossed' once or more, the age class is increased accordingly:

```
CREATE OR REPLACE FUNCTION tools.age_class(
  animal_id integer,
  acquisition_time timestamp with time zone)
RETURNS integer AS
$BODY$
DECLARE
  animal_age_class_code_capture integer;
  add_year integer;
  animal_date_capture date;
BEGIN
-- Retrieve the age class at first capture
  animal_age_class_code_capture = (SELECT age_class_code FROM main.animals
                                   WHERE animals_id = animal_id);

-- If the animal is already an adult then all locations will be adult
IF animal_age_class_code_capture = 3 THEN
  RETURN 3;
END IF;

-- In case the animal at capture was not an adult, the function checks if
the capture was before or after April.
-- In the second case, the age class will increase the April of the next
year.
  animal_date_capture = (SELECT age_class_code FROM main.animals
                         WHERE animals_id = animal_id);

IF EXTRACT(month FROM animal_date_capture) > 3 THEN
  add_year = 1;
ELSE
  add_year = 0;
END IF;
```

```
-- If the animal was an yearling at capture, the function checks if it went
through an age class increase.
IF animal_age_class_code_capture = 2 THEN
  IF acquisition_time > ((extract(year FROM animal_date_capture) +
                          add_year)|| '/4/1')::date THEN
    RETURN 3;
  ELSE
    RETURN 2;
  END IF;
END IF;
-- If the animal was a fawn at capture, the function checks if it went
through two and then one age class increase.
IF animal_age_class_code_capture = 1 THEN
  IF acquisition_time > ((extract(year FROM animal_date_capture) + add_year + 1)
                         || '/4/1')::date THEN
    RETURN 3;
  ELSEIF acquisition_time > ((extract(year FROM animal_date_capture) + add_year)
                            || '/4/1')::date THEN
    RETURN 2;
  ELSE
    RETURN 1;
  END IF;
END IF;

END;
 $BODY$
LANGUAGE plpgsql;

COMMENT ON FUNCTION tools.age_class(integer, timestamp with time zone)
IS 'This function returns the age class at the acquisition time of a location.
It has two input parameters: the id of the animal and the timestamp. According
to the age class at first capture, the function increases the class by 1 every
time the animal goes through a defined day of the year (1st April).';
```

Unfortunately, all the animals in the database are adults, so no change in the age class is possible. In any case, as an example of usage, we report the code to retrieve the dynamic age class:

```
SELECT
  animals_id,
  acquisition_time,
  tools.age_class(animals_id, acquisition_time)
FROM main.gps_data_animals
ORDER BY animals_id, acquisition_time
LIMIT 10;
```

The result is

```
 animals_id |    acquisition_time    | age_class
------------+------------------------+-----------
          1 | 2005-10-18 22:00:54+02 |         3
          1 | 2005-10-19 02:01:23+02 |         3
          1 | 2005-10-19 06:02:22+02 |         3
          1 | 2005-10-19 10:03:08+02 |         3
          1 | 2005-10-20 22:00:53+02 |         3
          1 | 2005-10-21 02:00:48+02 |         3
          1 | 2005-10-21 06:00:53+02 |         3
          1 | 2005-10-21 10:01:42+02 |         3
          1 | 2005-10-21 14:03:11+02 |         3
          1 | 2005-10-21 18:01:16+02 |         3
```

Generation of Random Points

In some cases, it can be useful to generate a determined number of random points in a given polygon (e.g. resource selection function, in order to get a representation of the available habitat). This can be done using the database function reported below. It requires a polygon (or multipolygon) geometry and the desired number of points as input. The output is the set of points:

```
CREATE OR REPLACE FUNCTION tools.randompoints(
  geom geometry,
  num_points integer,
  seed numeric DEFAULT NULL)
RETURNS SETOF geometry AS
$$
DECLARE
  pt geometry;
  xmin float8;
  xmax float8;
  ymin float8;
  ymax float8;
  xrange float8;
  yrange float8;
  srid int;
  count integer := 0;
  bcontains boolean := FALSE;
  gtype text;
BEGIN
SELECT ST_GeometryType(geom)
INTO gtype;

IF ( gtype != 'ST_Polygon' ) AND ( gtype != 'ST_MultiPolygon' ) THEN
  RAISE EXCEPTION 'Attempting to get random point in a non polygon geometry';
END IF;

SELECT ST_XMin(geom), ST_XMax(geom), ST_YMin(geom), ST_YMax(geom),
  ST_SRID(geom) INTO xmin, xmax, ymin, ymax, srid;
```

```
SELECT xmax - xmin, ymax - ymin
INTO xrange, yrange;

IF seed IS NOT NULL THEN
  PERFORM setseed(seed);
END IF;

WHILE count < num_points LOOP
  SELECT
    ST_SetSRID(ST_MakePoint(
      xmin + xrange * random(), ymin + yrange * random()), srid)
  INTO pt;
  SELECT ST_Contains(geom, pt)
  INTO bcontains;
  IF bcontains THEN
    count := count + 1;
    RETURN NEXT pt;
  END IF;
END LOOP;
RETURN;
END;
$$
LANGUAGE 'plpgsql';

COMMENT ON FUNCTION tools.randompoints(geometry, integer, numeric)
IS 'This function generates a set of random points into a given polygon (or
multipolygon). The number of points and the polygon must be provided as
input. A third optional parameter can define the seed, and thus generate a
consistent (random) set of points.';
```

It can be used in a view to generate a set of points automatically whenever the view is called. In this example, the study area is used as input geometry to generate 100 random points:

```
CREATE VIEW analysis.view_test_randompoints AS
  SELECT
    row_number() over() AS id,
    geom::geometry(point, 4326)
  FROM
    (SELECT
      tools.randompoints(
        (SELECT geom FROM env_data.study_area), 100)AS geom) a;
COMMENT ON VIEW analysis.view_test_randompoints
IS 'This view is a test that shows 100 random points (generated every time
that the view is called) into the boundaries of the first polygon stored in
the home_ranges_mcp table.';
```

The *row_number()* is added to generate a unique integer associated with each point; otherwise, some of the client applications will not be able to deal with this view. If you visualise the view in a GIS environment (e.g. in QGIS), you will notice that the set of points changes every time that you refresh your GIS interface. This is because the view generates a new set of points at every call. If you need to

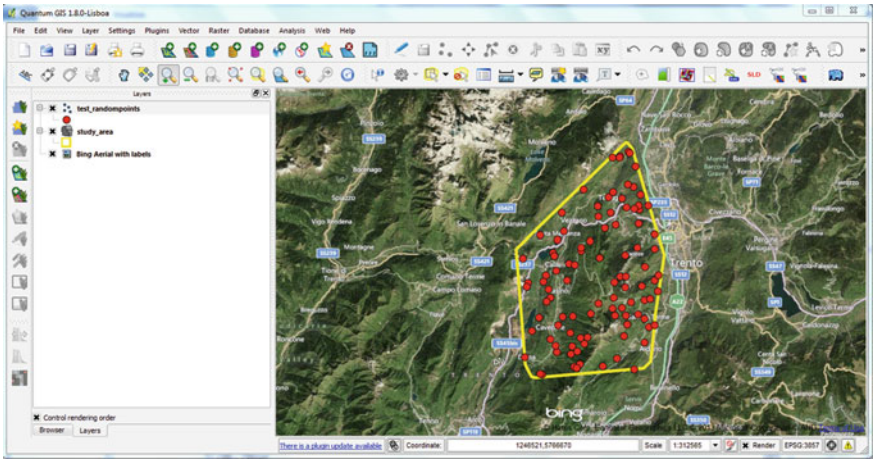

Fig. 9.4 Random points generated in a polygon

consistently generate the same set of points for reproducibility, you can specify a third parameter that defines the seed[9] (a numeric value in the range from -1 to 1) based on the PostgreSQL *setseed*[10] function. The seed option allows you to reproduce the same results while keeping the generation process random. Changing the seed will generate another set of random locations. Another option is to make the random points permanent and upload the result into a permanent table that can then be processed further (e.g. intersected with environmental layers):

```
CREATE TABLE analysis.test_randompoints AS
  SELECT
    row_number() over() AS id,
    geom::geometry(point, 4326)
  FROM
    (SELECT
      tools.randompoints(
        (SELECT geom FROM env_data.study_area), 100)AS geom) a;
ALTER TABLE analysis.test_randompoints
  ADD CONSTRAINT test_randompoints_pk PRIMARY KEY(id);

COMMENT ON TABLE analysis.test_randompoints
IS 'This table is a test that permanently stores 100 random points into the
boundaries of the first polygon stored in the home_ranges_mcp table.';
```

A graphical illustration of the result is illustrated in Fig. 9.4.

[9] http://en.wikipedia.org/wiki/Random_seed.

[10] http://www.postgresql.org/docs/9.2/static/sql-set.html.

References

Calenge C, Dray S, Royer-Carenzi M (2009) The concept of animals' trajectories from a data analysis perspective. Ecol Inform 4:34–41. doi:10.1016/j.ecoinf.2008.10.002

Horne JS, Garton EO, Krone SM, Lewis JS (2007) Analyzing animal movements using Brownian bridges. Ecology 88:2354–2363. doi:10.1890/06-0957.1

Kranstauber B, Kays R, LaPoint SD, Wikelski M, Safi K (2012) A dynamic Brownian bridge movement model to estimate utilization distributions for heterogeneous animal movement. J Anim Ecol 81:738–746. doi:10.1111/j.1365-2656.2012.01955.x