Ferdinando Urbano · Francesca Cagnacci
Editors

Spatial Database for GPS Wildlife Tracking Data

A Practical Guide to Creating a Data Management System with PostgreSQL/ PostGIS and R

Foreword by Ran Nathan

Springer

Spatial Database for GPS Wildlife Tracking Data

Ferdinando Urbano · Francesca Cagnacci
Editors

Spatial Database for GPS Wildlife Tracking Data

A Practical Guide to Creating a Data Management System with PostgreSQL/PostGIS and R

Foreword by Ran Nathan

Springer

Editors
Ferdinando Urbano
Università Iuav di Venezia
Venice
Italy

Francesca Cagnacci
Research and Innovation Centre
Fondazione Edmund Mach
San Michele all'Adige,
Trento
Italy

Additional material to this book can be downloaded from http://extras.springer.com

ISBN 978-3-319-03742-4 ISBN 978-3-319-03743-1 (eBook)
DOI 10.1007/978-3-319-03743-1
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014930228

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Extra Materials on http://extras.springer.com

Test data sets

trackingDB_datasets.zip	Test data sets used as reference in the exercises presented in the book. These include: GPS locations (from 5 GPS sensors); activity data (from 1 activity sensor); animals; sensors; deployment of sensors on animals; environmental layers in vector format (limits of the study area, administrative boundaries, meteorological stations, roads); environmental layers in raster format (land cover, digital elevation model, normalized difference vegetation index).

Code

trackingDB_code.zip	Plain text files with the commented code (database and R) presented in Chapters 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 (one .txt file per chapter). The code can be easily copy/pasted by readers to facilitate the creation of their own database.

Dump of the database

trackingDB_dump.zip	Plain SQL dump of the database built in Chapters 2, 3, 4, 5, 6, 7, 8 and 9. This allows users to skip the steps to create the database as presented through chapters, thus providing the final outcome. This is needed for readers who buy individual chapters (the code presented in each chapter makes reference to the database created sequentially in the previous chapters).

Foreword

A group of kindergarten kids are walking in a valley with their teacher. Children, especially at this early age, have keen eyes and unbounded imagination and curiosity. They commonly discover interesting objects well before adults do. One girl discovers an unusual stone and excitedly calls the others to see. The crowd of kids circles around the object, offering various interpretations. 'I can see this stone has lots of iron in it', says a future geologist. 'This was left here by ancient people', says a boy from a family with a long tradition in the valley. 'It's a 'metarite' that came from the sky', says the dreamer. 'Meteorite, not metarite' corrects the teacher, and gets closer to figure out what this odd stone is. Suddenly, the 'stone' jumps. 'It's alive!' shout the kids together in surprise, running after the neatly camouflaged grasshopper as it jumps to escape the crowd.

The movement of organisms, the subject of this book, is the most fundamental characteristic of life. Most, if not all, species—microorganisms, plants, and animals—move at one or more stages of their life cycle. These movements affect individual fitness, determine major evolutionary pathways and trait selection, and shape key ecological processes, including some of the most stressful global environmental and health problems we face today. Humans, including such notable figures as Aristotle, Heraclitus, Jeremiah, and Lao-tzu, have long recognized the movement of organisms as a fundamental feature of nature and human life. Consequently, movement research has been rich and insightful since the early days of science, with remarkable contributions from such luminaries as Newton, Darwin, and Einstein. Despite this, progress in movement research has long been hampered by the immense practical difficulties involved in collecting and analysing movement data. Furthermore, movement research has long been scattered among numerous subfields, and until recently lacked a unifying framework to advance our understanding of all movements types and all organisms.

These obstacles are now being rapidly overcome. We are currently seeing a golden age of unprecedented progress and discovery in movement research. Three main processes are responsible for this rapid advancement: (1) the development of new tracking techniques to quantify movement (Tomkiewicz et al. 2010), (2) powerful data analysis tools to examine key questions about movement and its causes and consequences (Schick et al. 2008), and (3) conceptual and theoretical frameworks for integrating movement research (Nathan et al. 2008). Many new tools have been developed recently to record the movement of various organisms

with levels of accuracy, frequency, and duration that are much higher than they were just a few years ago. This has strongly improved our capacity to track insects, fish, and other relatively small organisms. However, arguably the most dramatic development in animal biotelemetry is driven by GPS-based devices, which have been applied mostly to track mammals and birds of relatively large (>100 g) body mass. Although such devices remain limited to a small proportion of the world's biota, they are now applicable to a large and increasing number of species, and have caused a dramatic shift of movement ecology from a data-poor to a data-rich research enterprise. The parallel development of bio-logging techniques that use accelerometers and various other sensors, as well as GIS and remote sensing techniques, have further enriched our ability to quantify the internal state, behaviour, and the external environment of the animal *en route*. Therefore, this book's focus on animal-borne GPS data is important for the advancement of the most active frontiers of movement research.

A recent prognosis of the state of the art of physics, as an example for scientific research more generally, has concluded that we are lucky to live in an era when both tools and ideas are getting strong (Dyson 2012). The same applies to movement research. Along these lines, the three main processes that drive the current rapid advancement of movement ecology should be strongly interlinked. This is precisely the contribution that this book attempts to make. In order to efficiently utilize the large volume of high-quality GPS data, to take advantage of the power of advanced data analysis tools, and to facilitate the investigation of key hypotheses and the promotion of integrative theoretical frameworks, we first need to properly manage the data sets. Although this requirement holds for data sets of all sizes, and is therefore relevant to all movement ecology projects, a key challenge is to manage the large and rapidly growing GPS tracking data sets that have been collected by many groups studying diverse species across different ecosystems. This marks the entrance of movement ecology to the Big Data era, which necessitates revolutionary improvements in data management and data analysis techniques, and resembles the bioinformatics revolution of genomics and proteomics in the previous decade (Mount 2004). Ecologists have been late to enter the information age, and will need to conquer the strong traditions that have developed over the long data-poor history of this discipline in order to utilize the huge scientific opportunities of the Big Data era (Hampton et al. 2013). The integrative, transdisciplinary approach of movement ecology facilitates this task, and this book is the most comprehensive attempt to date to guide movement ecologists on how to deal with their data sets.

The data management challenge involves multiple tasks, including gathering, processing, storing, searching, organizing, visualizing, documenting, analysing, retrieving, and sharing the data. Furthermore, the basic features of the data sets—the data type, the spatial and temporal resolution, the errors, and the biases—tend to vary across studies, which has created the need for general guidelines on how to standardize data management in a consistent and robust manner. Furthermore, the growing availability of complementary quantitative data, both on the internal state of the focal animal and its biotic and abiotic environment (e.g. Dodge et al. 2013)

has increased our ability to link movement to fitness, elucidate its proximate underlying mechanisms, and explore social and other biotic interactions, the impact of humans on animals, and their response to environmental changes. Although this progress is desirable, it also amplifies and complicates the Big Data problem, requiring coordination among different tools, spatial databases, and approaches. These challenges can be overwhelming, and the current tendency of individual researchers and research groups to reinvent the wheel with their own data management tools and protocols should be replaced with generic methods developed through fruitful joint efforts of the scientific community. Ferdinando Urbano, Francesca Cagnacci, and their colleagues have performed a tremendous service in this regard by taking the first steps in this long journey, addressing some of the major challenges of managing movement data sets in a comprehensive and useful way.

The book starts with a useful Introduction and includes two theoretical/review chapters, Chaps. 1 and 13. All other chapters provide clear technical guidelines on how to create and automatically organize a GPS database, how to integrate ancillary information and manage errors, and how to represent and analyse GPS tracking data, as well as a glimpse into complementary data from other bio-logged sensors. The practical guidelines are implemented in SQL codes for the open source database management system PostgreSQL and its spatial extension PostGIS—the software platform recommended here for GPS data sets—as well as R codes for statistical analysis and graphics. Each technical chapter also includes hands-on examples of real-life database management tasks based on freely available data sets, given in a sequential manner to gradually guide the learning process. The chapters cover a wide range of topics, avoiding the need to provide an in-depth review and illustration of all state-of-the-art methods and tools; this is a logical and welcoming strategy given the lack of a basic textbook on this subject. The book is targeted at biologists, ecologists, and managers working on wildlife (GPS) tracking data, with special emphasis on wildlife management, conservation projects and environmental planning. It also provides an excellent reference for the target groups to establish collaborations with ICT specialists.

I am grateful to this excellent team of authors for sharing their experience and insights on the critically important technical building blocks of data management for animal movement research. I am particularly happy to see that such a methodological approach is not presented in a ‘dry’ technical manner, but rather in the light of key research questions within the broader theoretical background of movement ecology, in a user-friendly manner, and with clear links to real-life applications. The authors should be warmly congratulated for producing this important contribution. This book will provide an important pioneering impetus for a large and successful research enterprise of a Big Movement Ecology Data era, which some research groups have already experienced and many others are likely to experience in the near future.

Jerusalem, December 2013 Ran Nathan

References

Dodge S, Bohrer G, Weinzierl R, Davidson SC, Kays R, Douglas D, Cruz S, Han J, Brandes D, Wikelski M (2013) The environmental-data automated track annotation (Env-DATA) system: linking animal tracks with environmental data. Mov Ecol 1:3

Dyson FJ (2012) Is science mostly driven by ideas or by tools? Science 338:1426–1427

Hampton SE, Strasser CA, Tewksbury JJ, Gram WK, Budden AE, Batcheller AL, Duke CS, Porter JH (2013) Big data and the future of ecology. Front Ecol Environ 11:156–162

Mount DW (2004) Bioinformatics: sequence and genome analysis. Cold Spring Harbor Laboratory Press, New York

Nathan R, Getz WM, Revilla E, Holyoak M, Kadmon R, Saltz D, Smouse PE (2008) A movement ecology paradigm for unifying organismal movement research. Proc Natl Acad Sci USA 105:19052–19059

Schick RS, Loarie SR, Colchero F, Best BD, Boustany A, Conde DA, Halpin PN, Joppa LN, McClellan CM, Clark JS (2008) Understanding movement data and movement processes: current and emerging directions. Ecol Lett 11:1338–1350

Tomkiewicz SM, Fuller MR, Kie JG, Bates KK (2010) Global positioning system and associated technologies in animal behaviour and ecological research. Philos Trans R Soc B 365:2163–2176

Acknowledgments

This book is largely based on the lessons of the summer school 'Next Generation Data Management in Movement Ecology', held in Berlin, Germany on 3–7 September 2012, and organized by the Leibniz Institute for Zoo and Wildlife Research (IZW, Germany), the Swedish University of Agriculture Sciences (SLU, Sweden), the Edmund Mach Foundation (FEM, Italy), and the Max Planck Institute for Ornithology (MPIO, Germany). These lessons synthesize the knowledge developed over continuous refinement of wildlife tracking data management supported by several projects related to movement ecology at Fondazione Edmund Mach (Trento, Italy), Vectronic Aerospace GmbH (Berlin, Germany), the EURODEER cooperative project (http://www.eurodeer.org), the Norwegian Institute for Nature Research (Trondeim, Norway), and the Leibniz Institute for Zoo and Wildlife Research (Berlin, Germany).

The authors are also particularly grateful to the following institutions for directly or indirectly supporting their contributions to the book: the Swedish University of Agricultural Sciences (Umeå, Sweden), the Max Planck Institute for Ornithology (Radolfzell, Germany), the University of Konstanz (Konstanz, Germany), the University of Florida (Fort Lauderdale, FL, USA), the German Science Foundation (Germany), and the U.S. National Science Foundation (USA).

The ideas proposed here also benefited from the contribution of many scientists and technologists through fruitful discussion, in particular Clément Calenge, Maria Damiani, Ralf Hartmut Güting, Manuela Panzacchi, and Veit Zimmermann.

We warmly thank Roger Bivand, Mark Hebblewhite, Peter Klimley, Max Kröschel, George MacKerron, and Philip Sandstrom for their invaluable suggestions on previous versions of the text.

A special acknowledgment to our co-authors Mathieu Basille, who gave a priceless scientific and technical contribution to the whole book, and Sarah Cain Davidson, who helped keep the overall coherence across chapters.

Contents

Introduction

Francesca Cagnacci, Mathieu Basille, Anne Berger, Sarah Cain Davidson, Holger Dettki, Bram van Moorter and Ferdinando Urbano

F. Cagnacci
Biodiversity and Molecular Ecology Department, Research and Innovation Centre, Fondazione Edmund Mach, via E. Mach 1, 38010 S.Michele all'Adige, TN, Italy
e-mail: francesca.cagnacci@fmach.it

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

A. Berger
Leibniz Institute for Zoo and Wildlife Research, Alfred-Kowalke-Straße 17, D-10315 Berlin, Germany
e-mail: berger@izw-berlin.de

S. C. Davidson
Max Planck Institute for Ornithology, Am Obstberg 1, 78315 Radolfzell Germany and Department of Civil, Environmental and Geodetic Engineering, The Ohio State University, 2070 Neil Ave, Columbus, OH 43210, USA
e-mail: sdavidson@orn.mpg.de

H. Dettki
Umeå Center for Wireless Remote Animal Monitoring (UC-WRAM), Department of Wildlife, Fish, and Environmental Studies, Swedish University of Agricultural Sciences (SLU), Skogsmarksgränd, SE-901 83 Umeå, Sweden
e-mail: holger.dettki@slu.se

B. van Moorter
Norwegian Institute for Nature Research (NINA), Høgskoleringen 9, 7034 Trondheim, Norway
e-mail: Bram.van.moorter@gmail.com

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

The Long and Winding Road to Movement Ecology: Let's put Things in Order, First

Animal movement is a proximate response to local environmental conditions, such as climate, chemophysical parameters, resources, and the presence of mates or predators; in other words, the context in which animals are born, survive, reproduce, and die. At the same time, movement is the result of complex evolutionary mechanisms, uniquely blending gene expression into physiological and behavioural responses. Animal movement has been described as the glue that ties ecological processes together (Turchin 1998; Nathan et al. 2008) and as an important mechanistic link between ecology and evolution (Cagnacci et al. 2010). Moreover, the increasing concern for rapidly changing ecosystems, under climate and land use change, brings with it an urgent and heightened interest in the capacity of animals to respond to such changes. An exciting perspective offered by animal movement is the possibility to understand this phenomenon at different spatial scales and levels of organisation, from individuals, to populations over landscapes and to the distribution range of species (Mueller and Fagan 2008). Such stimulating theoretical framework and widely relevant applied perspective clearly justify the ever-increasing interest in animal tracking studies, especially those based on animal-borne data sets, i.e. those obtained through the deployment of tracking units on individual animals. The innovation in Global Positioning System (GPS) technology, combined with systems for remote data transfer, has particularly favored GPS-based animal telemetry to become a standard in wildlife tracking (Cagnacci et al. 2010).

The advancement of a movement ecology theoretical framework has been paralleled by technological progress that allows ecologists to obtain a huge amount and diversity of empirical animal movement data sets. However, this fast-growing and expanding process has not been followed by an equally rapid development of procedures to manage and integrate animal movement data sets, thus leaving a gap between the acquisition of data and the overarching scientific questions they have the potential to address (Urbano et al. 2010).

The ideal objective of any movement ecology study is rooted in relevant ecological questions that can contribute to theory and inform conservation and management actions. For example, a study on natal dispersal can help in identifying barriers to gene flow, or an analysis on environmental characteristics affecting population performance can support decisions on protection areas and conservation corridors. Whatever the question, the second necessary step should be the evaluation of the appropriate methodology, and specifically deciding whether individual marking with Global Navigation Satellite Systems (GNSS) devices such as GPS (or other sensors) is the most effective and informative approach to pursuing the final goal (see discussion in Hebblewhite and Haydon 2010). If the answer is yes, GPS units should be ideally deployed according to an *a priori* sampling design (e.g. number, sex ratio and spacing of sampled individuals) and relevant information on marked individuals should be collected (e.g. age and body mass),

as well as on the population of reference and the environmental context (e.g. population density, presence of competitor species/predators, human activity pressure). Collection of GPS locations of marked animals should finally flow, and can be complemented with environmental parameters (e.g. weather, habitat types and vegetation indexes based on remote sensing).

The reason for this book comes at this stage of the scientific method: what to do with these data? How to handle, manage, store and retrieve them, and how to eventually feed them to analysis tools such as statistics packages or Geographic Information Systems (GIS) and test scientific hypotheses? These operations, which might be assumed to be secondary compared to the overarching goal of answering scientific questions, can instead become overwhelming and hamper the efficiency and consistency of the whole process. Animal ecologists, wildlife biologists and managers, to whom this book is mainly addressed, are rarely exposed to the basics of computer science in their training, and may select common tools such as spreadsheets, or operate in a 'flat-files' fashion. However, the quantity and complexity of GPS and other bio-logged data require a proper software architecture to be fully exploited and not wasted or, worse, misused. This book is a guide to manage and process wildlife tracking data with an advanced software platform based on a spatial database. It is neither a manual on database programming nor on wildlife tracking; instead, it aims to fill the gap between the state-of-the-art knowledge on data management and its application to wildlife tracking data. This problem-solving oriented guide focuses on how to extract the most from GPS animal tracking data, while preventing error propagation and optimizing the performance of analysis. Using databases to manage tracking data implies a considerable initial effort for those who are not familiar with these tools; however, the time spent learning will pay off in time saved for the management and processing of the data, and in the quality of the analysis and final results. Moreover, once a consistent database structure is built to code the storing and management of data, more familiar tools can be used as interfaces.

Another important advantage of using a structured and consistent software platform for management of wildlife tracking data is the ever-increasing importance of cooperative projects and data sharing. Deploying sensors on animals can be expensive, both in terms of capture logistics and actual cost of tracking devices, and implies some amount of stress for the marked animals. These costs pose financial and even moral incentives to maximize the effective use of animal-borne data. Moreover, many large-scale questions, such as evaluating the effects of climate and land use change on animal distributions, can be properly addressed only by studying multiple populations, or by integrating data from several species or time periods. Data requirements for such studies are virtually impossible for any single research institution to meet, and can only be achieved by cooperative research and data sharing (Whitlock 2011). Spatial databases are essential tools to assure data standards and interoperability, along with a safe multiple users operational environment.

In this manual, a sequential set of exercises guides readers step by step through setting up a spatial database to manage GPS tracking data together with ancillary

Fig. 1 F10, one of the sample animals studied in this book: an adult female roe deer (*Capreolus capreolus*), wearing a Vectronic Aerospace GmbH GPS-Plus yellow collar

information (e.g. characteristics of animals and sensors), environmental layers (e.g. land cover, altitude and vegetation index), and data from other bio-logged sensors (e.g. accelerometers). Data from five GPS and one activity sensors deployed on roe deer (*Capreolus capreolus*; Fig. 1) in the Italian Alps (Fig. 2) are used as the test data set[1]. In each chapter, data management problems are contextualized in relation to issues that scientists and wildlife managers face when they work with tracking data. The goal of the book is to illustrate conceptual and technical solutions and to offer a practical example of general validity for most of the animal species target of movement ecology studies.

The world of animal-borne telemetry, or bio-logging, comprises information not limited to location data of marked animals. The first natural extension is 3D coordinates, where two-dimensional coordinates (i.e. latitude and longitude) are complemented by altitude for avian species or depth for marine species. Other increasingly used parameters are triaxial acceleration (measured by accelerometers) and magnetic bearing (measured by magnetometers), which can integrate the movement trajectories with information on the type of body movement and therefore activity of animals. A diversity of other variables can be measured by bio-logging devices, providing information on the internal state of animals (biomechanics, energetics), and also their external state, i.e. the environment (pressure, chemophysical parameters) (see Ropert-Coudert and Wilson 2005 for an

[1] The test data set (trackingDB_datasets.zip) is available at http://extras.springer.com.

Fig. 2 Typical habitat characterizing the alpine area where the sample animals were tracked by means of GPS collars: open alpine pastures and mixed woodlands

early review on this topic). When justified by the scientific questions underpinning the empirical study, the use of multi-sensor platforms deployed on individual animals can be a powerful tool to obtain a complex and diversified picture of the animal in its environment. With the miniaturization of technology and adaptation of devices to an increasing number of species, multi-sensor platforms and their resulting data sets are likely to be even more common in the future (see, for example, deployment of miniature video cameras on birds, Rutz and Troscianko 2013). The presence of multiple sources of information (i.e. the sensors) fitted on the same animal does not represent a challenge *per se*, if each type of information is consistently linked to the animal and integrated with the other data sources. However, this requires a dedicated and consistent management structure, an expansion of the database requirements for tracking data alone. Although this book mainly deals with spatial data obtained from individual animals tracked with GPS, examples are provided to show how most of the conceptual background can be exported to other sensors or to multi-sensor platforms.

PostgreSQL and its spatial extension PostGIS are the proposed software platform to build the wildlife tracking database. This spatial database will allow management of virtually any volume of data from wildlife GPS tracking in a unique repository, while assuring data integrity and consistency, avoidance of error propagation, and limiting data duplication and redundancy. Moreover, this software platform offers the opportunity of automation of otherwise very time-consuming processes, such as, for example, the association between GPS

locations, animals and the environmental variables, and high performance of data retrieval, which can become cumbersome when data sets are very large. Databases are designed for being remotely accessed and used concurrently by multiple users, assuring data integrity and security (i.e. ability to define a diversified access policy). Last but not least, once the storing and management of data is coded in a robust database, data can still be visualized and accessed with common tools such as office packages (e.g. spreadsheets, database front-ends) or GIS desktop. Therefore, after the initial effort of designing the structure and populating the database, most users should be able to easily access the data.

The proposed SQL code can be directly used to create an operational test database and, at a later stage, as a template to create a more customized data management system. R is used as the reference statistical environment. The examples, illustrated in a MS Windows environment, can be easily adapted to the commonly used operating systems, such as Mas OS X and Linux, which are all supported by both PostgreSQL/PostGIS and R.

Chapter 1 is a review of the opportunities and challenges that are posed by GPS tracking from a data management perspective, with a brief introduction to (spatial) databases, which are proposed as the best available tool for wildlife tracking data management. Chapter 2 guides readers through the initial creation of a (PostgreSQL) database to store GPS data. Chapter 3 shows how to integrate ancillary information on both animals and GPS devices. Chapter 4 presents a solution to associating GPS positions to animals in an automated fashion. Chapter 5 explores the features offered by PostGIS, the spatial extension of PostgreSQL, transforming the coordinates received from the GPS sensors into 'spatial objects', i.e. points embedding spatial characteristics. Chapter 6 uses the spatial tools of the database to add a set of spatial ancillary information in the form of vectors (points, lines, and polygons) and rasters and to connect these data sets with GPS tracking data. Chapter 7 focuses on the integration of ancillary information that captures the temporal variability of the environment, with a practical example based on the remote sensing image time series of Normalized Difference Vegetation Index (NDVI), a proxy of vegetation biomass. Chapter 8 is dedicated to the detection and management of erroneous location and potential outliers. Chapter 9 introduces methods for representing and analysing tracking data (e.g. trajectories, home range polygons, probability surfaces) using database functions. Chapter 10 explains how to connect R, a powerful software environment for statistical computing and graphics, with the database and describes a set of typical tracking data analyses performed with R. Chapter 11 introduces a more direct and efficient approach to integrate R and PostgreSQL: Pl/R. Chapter 12 discusses the integration of GPS tracking data with other bio-logged sensors, with a practical example based on accelerometers. Chapter 13 gives an overview of data standards and software interoperability, with special reference to data sharing.

Apart from Chaps. 1, 12 and 13, which introduce more general and theoretical topics, each chapter proposes exercises that are developed sequentially. Although it is possible to perform an exercise from a later chapter on its own by restoring the

dump of the whole database[2], we recommend completing the exercises in the order in which they are presented in the book to have a full understanding of the whole process. We also suggest acquiring some general understanding of databases and spatial databases before reading this book.

References

Cagnacci F, Boitani L, Powell RA, Boyce MS (2010) Animal ecology meets GPS-based radiotelemetry: a perfect storm of opportunities and challenges. Philos Trans R Soc B 365:2157–2162

Hebblewhite M, Haydon DT (2010) Distinguishing technology from biology: a critical review of the use of GPS telemetry data in ecology. Philos Trans R Soc B 365:2303–2312

Mueller T, Fagan WF (2008) Search and navigation in dynamic environments—from individual behaviours to population distributions. Oikos 117:654–664

Nathan R, Getz WM, Revilla E, Holyoak M, Kadmon R, Saltz D, Smouse PE (2008) A movement ecology paradigm for unifying organismal movement research. Proc Natl Acad Sci U.S.A. 105:19052–19059

Ropert-Coudert Y, Wilson R (2005) Trends and perspectives in animal-attached remote sensing. Front Ecol Environ 3(8):437–444

Rutz C, Troscianko J (2013) Programmable, miniature video-loggers for deployment on wild birds and other wildlife. Methods Ecol Evol 4:114–122

Turchin P (1998) Quantitative analysis of movement: measuring and modeling population redistribution in animals and plants. Sinauer Associates, Inc., Sunderland

Urbano F, Cagnacci F, Calenge C, Cameron A, Neteler M (2010) Wildlife tracking data management: a new vision. Philos Trans R Soc B 365:2177–2186

Whitlock MC (2011) Data archiving in ecology and evolution—best practices. Trends Ecol Evol 26(2):61–65. doi:10.1016/j.tree.2010.11.006

[2] The dump (trackingDB_dump.zip) is available at http://extras.springer.com.

Chapter 1
Wildlife Tracking Data Management: Chances Come from Difficulties

Holger Dettki, Ferdinando Urbano, Mathieu Basille and Francesca Cagnacci

Abstract In recent years, new wildlife tracking and telemetry technologies have become available, leading to substantial growth in the volume of wildlife tracking data. In the future, one can expect an almost exponential increase in collected data as new sensors are integrated into current tracking systems. A crucial limitation for efficient use of telemetry data is a lack of infrastructure to collect, store and efficiently share the information. Large data sets generated by wildlife tracking equipment pose a number of challenges: to cope with this amount of data, a specific data management approach is needed, one designed to deal with data scalability, automatic data acquisition, long-term storage, efficient data retrieval, management of spatial and temporal information, multi-user support and data sharing and dissemination. The state-of-the-art technology to meet these challenges are relational database management systems (DBMSs), with their dedicated spatial extension. DBMSs are efficient, industry-standard tools for storage, fast retrieval and manipulation of large data sets, as well as data dissemination to client

H. Dettki (✉)
Umeå Center for Wireless Remote Animal Monitoring (UC-WRAM),
Department of Wildlife, Fish, and Environmental Studies,
Swedish University of Agricultural Sciences (SLU), Skogsmarksgränd,
SE-901 83 Umeå, Sweden
e-mail: holger.dettki@slu.se

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

F. Cagnacci
Biodiversity and Molecular Ecology Department, Research and Innovation Centre,
Fondazione Edmund Mach, via E. Mach 1, 38010 S.Michele all'Adige, TN, Italy
e-mail: francesca.cagnacci@fmach.it

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_1,

programs or Web interfaces. In the future, we expect the development of tools able to deal at the same time with both spatial and temporal dimensions of animal movement data, such as spatiotemporal databases.

Keywords GPS tracking · Large data set · Database management system · Spatial database

Introduction

In recent years, new wildlife tracking and telemetry technologies have become available, allowing remote data capture from a steadily increasing number of taxa, species and individual animals. This has resulted in a substantial increase in the volume of data gathered by researchers, environmental monitoring programs and public agencies. In the future, one can expect an almost exponential increase in collected data as new sensors, e.g. to monitor health status, interactions among individuals, or other animal-centred variables, are integrated into current biologging systems on animals. Data can be remotely transferred to operators (e.g. using Global System for Mobile Communications (GSM) networks or satellite systems such as Argos, Globalstar and Iridium), making near real-time monitoring of animals possible. Furthermore, positional information can now be complemented with a wide range of other information about the animals' environment made available by satellite remote sensing, meteorological models and other environmental observation systems.

The information embedded in animal-borne data sets is enormous and could be made available in a wider societal context than wildlife research or management. However, there is still a lack of suitable infrastructures to collect, store and efficiently share these data. In this chapter, and in the rest of the book, we offer a solution for a subset of animal-borne information, i.e. wildlife tracking data. With this term, we mainly refer to Global Positioning System (GPS)-based radiotelemetry (Cagnacci et al. 2010; Tomkiewicz et al. 2010). 'GPS' is here a synonym for all different existing or upcoming global navigation satellite system (GNSS)-based animal tracking systems. Most of the concepts and tools proposed in this book, however are also valid for other tracking systems (e.g. very high frequency (VHF) telemetry, radio frequency identification (RFID) systems, echolocation).

In the past, software tools for wildlife tracking studies were mainly developed on the basis of VHF radiotracking data, which are characterised by small and discontinuous data sets, and were focused on data analysis rather than data management (Urbano et al. 2010). Spatial data, such as animal locations and home ranges, were traditionally stored locally in flat files, accessible to a single user at a time and analysed by a number of independent applications without any common standards for interoperability. Now that GPS-based tracking data series have become the main reference, data management practices face new challenges.

As we discuss below, GPS telemetry usually provides locations separated by constant and short time intervals (varying from a few minutes to several hours) that accumulate in large data series. Thus, data should be securely, consistently and efficiently managed in order to minimise errors, increase the reliability and reproducibility of inferences and ensure data persistence (e.g. access to data on multiple occasions and by several persons). Further, there is an increasing call for sharing and distributing data to the global research community, for principle and opportunity, and wildlife tracking data are no exception. Indeed, deploying tracking devices and bio-logging sensors on wildlife is costly, and most projects are local or rely on limited sample sizes. Hence, to realise the full potential of locally collected data, researchers must be able to share with and distribute their data to the global research community (see Chap. 13).

Below, we summarise the requirements that wildlife tracking data represent in terms of data management, and opportunities offered by potential solutions. This analysis is largely drawn by Urbano et al. (2010), with updated considerations.

Requirements

The methodological approach and software architecture for managing wildlife tracking data have to meet the specific requirements of spatiotemporal data series which are the result of individual animals' behaviour. Thus, the first step is the definition of both data and marked animals' characteristics, as well as the users' needs.

- **Scalability:** GPS-based devices can currently record thousands of locations per animal over short time intervals (hours, days, months). The number of monitored individuals and species has steadily increased in recent years, due in part to decreases in costs, decreases in device size and availability of a growing range of device models. Data collected by additional bio-logging sensors can vastly increase the total amount of data collected. Data management methods must be able to accommodate this growing volume of data.
- **Periodic and automatic data acquisition:** Automated procedures to receive, process and store data from GPS telemetry devices are required when a near real-time data transfer system is provided by the tracking units.
- **Long-term storage for data reuse:** Data must be consistently stored and properly documented beyond the period of data collection and analysis to permit data archiving, reuse and sharing.
- **Efficient data retrieval:** As the data sets increase in size, effective data analysis depends on efficient data retrieval tools.
- **Management of spatial information:** GPS data are by definition spatiotemporal data (i.e. usually they represent moving objects). Retrieval, manipulation and management tools should then be specific to the spatial data domain.

- **Global spatial and time references:** The use of global time and spatial reference systems enables comparison with data sets from different regions and at different scales.
- **Heterogeneity of applications:** The complex nature of movement ecology requires that sensor data are visualised, explored and analysed by a wide range of task-oriented applications; therefore, the software architecture should support the integration of different software tools.
- **Integration of additional data sources:** Animal locations can be enhanced by other spatial (e.g. remote sensing, socioeconomic) and non-spatial information (e.g. capture details or life-history traits), as well as data from other bio-logging sensors; multiple spatial and non-spatial data sets should be correctly managed and efficiently integrated into a comprehensive data structure.
- **Multi-user support:** Wildlife tracking data sets are of interest to researchers, but also to a range of stakeholders, including for example public institutions (wildlife management offices, national parks), and private organisations (environmental groups, hunters). These users might need to access data simultaneously, both locally and remotely, with different access privileges.
- **Data sharing:** There is an increasing call for sharing data publicly or among research groups. This is discussed in more detail in Chap. 13. Technically, data sharing requires adherence to standard data formats, definition of metadata and methods for data storage and management that, in turn, guarantee interoperability.
- **Data dissemination/outreach:** Dissemination of data to the scientific community or outreach activities targeting the general public is important to supporting management decisions, fundraising and promoting a larger awareness of issues related to ecosystem changes and resilience to changes. This requires the integration of specific tools to visualise and make data accessible (e.g. Web-based data interfaces, mapping tools, or search engines).
- **Cost-effectiveness:** By choosing cost-effective software tools that can meet the above requirements, funding can be focused on the collection and analysis of data, rather than on data management.

Chances

All of these requirements must be satisfied to take full advantage of the information that wildlife tracking devices can provide. As the volume and complexity of these data sets increase, the software tools used in the past by most animal tracking researchers are not sustainable, and thus there is an urgent need to adopt new software architectures.

Fortunately, software solutions exist and have a large user base. The reference solutions for data management are relational or object-relational database management systems (DBMSs), with their dedicated spatial extensions. DBMSs are efficient tools for storage, fast retrieval and manipulation of data (Urbano et al. 2010).

From a strictly technical point of view, advantages of DBMSs for tracking and movement ecology studies include the following:

- **Storage capacity:** Virtually any potential volume of data from wildlife GPS tracking or other sensor data can be stored in a DBMS.
- **Data integrity:** Data entry, changes and deletions can be checked to comply with specific rules.
- **Data consistency:** DBMSs fully support reversible transactions and transaction logging to ensure traceability of data operations and proper data management.
- **Automation of processes:** DBMSs can be empowered by defining internal functions and triggers; thus, a wide range of routinely complex work procedures can be automatically and efficiently performed inside the database itself.
- **Data retrieval performance:** The use of indexes effectively decreases querying time.
- **Management of temporal data types:** Time zones or daylight saving settings linked to temporal data types are supported and allow time consistency across study areas and times of year.
- **Reduced data redundancy:** The use of primary keys avoids data replication and the adoption of a normalized relational data model reduces data redundancy.
- **Client/server architecture:** Advanced DBMSs provide data through a central service, to which many applications can be connected and used as database front-end clients.
- **Advanced exploratory data analysis:** Data mining techniques for automatic knowledge discovery of information embedded in large spatial data sets must be applied in consistent and structured environments such as DBMSs.
- **Data models:** Data models are the logical core of DBMSs and allow linking and integration of data sources by means of complex relationships; this is not only necessary for consistently structuring the database, but is also an extremely useful way to force users to clarify the ecological/biological relational links between groups of data. This will be discussed extensively in Chaps. 2, 3 and 4.
- **Multi-user environment:** Data can be accessed by multiple users at the same time, keeping control on the coherence between operations performed by them, and maintaining a structured data access policy (see below).
- **Data security:** A wide range of data access controls can be implemented, where each user is constrained to the use of specific sets of operations on defined subsets of data.
- **Standards:** Consolidated industry standards for databases, data structure and metadata facilitate interoperability with client applications and data sharing among different research groups (see Chap. 13).
- **Backup and recovery:** Regular backup and potential disaster recovery processes can be efficiently managed.
- **Cost-effectiveness:** Multiple open source DBMSs software solutions are available that have large user and development communities, as well as extensive free and low-cost resources for training and support.

Spatial and Spatiotemporal Extensions

In addition to the important features listed above, spatial tools are increasingly integrated within databases that now accommodate native spatial data types (e.g. points, lines, polygons, rasters). These spatial DBMSs are designed to store, query and manipulate spatial data, including spatial reference systems. In a spatial database, spatial data types can be manipulated by a spatial extension of the Structured Query Language (SQL), where complex spatial queries can be generated and optimised with specific spatial indexes. Today, all major DBMS providers offer native spatial capabilities and functions in their products.

Spatial databases can easily be integrated with Geographical Information System (GIS) software, which can be used as client applications. Further, few desktop GIS are optimised for managing large vector data sets and complex data structures. Spatial databases, instead, are the tool of choice for performing simple spatial operations on a large set of elements. Thus, simple but massive operations on raw data can be preprocessed within the spatial database itself, while more advanced spatial analysis on subsequent data sets can rely on GIS and the spatial statistics packages connected to it.

A further promising extension to spatial data models is the adoption of spatiotemporal data models (e.g. Kim et al. 2000; Pelekis et al. 2004; Güting and Schneider 2005). In these models, locations are characterised by both a spatial and a temporal dimension that are combined into one unique, double-faced attribute of movement. Spatiotemporal databases will extend the spatial data model for animals by integrating data types and functions specifically related to the spatiotemporal nature of animal movements (e.g. considering 'movement' as an attribute of the animal instead of relying on clusters of location objects with timestamps). This approach would help to decipher the relationships between animal movement, habitat use and environmental conditions. Although commonly used DBMSs do not yet support an integrated spatiotemporal extension, spatiotemporal databases (e.g. SECONDO[1], Güting et al. 2004), which are undergoing intense development, will be the natural evolution for wildlife tracking data management tools in the future.

References

Cagnacci F, Boitani L, Powell RA, Boyce MS (2010) Animal ecology meets GPS-based radiotelemetry: a perfect storm of opportunities and challenges. Philos Trans R Soc B 365:2157–2163. doi:10.1098/rstb.2010.0107

Güting RH, Behr T, de Almeida VT, Ding Z, Hoffmann F, Spiekermann M (2004) SECONDO: an extensible DBMS architecture and prototype. Fernuniversität Hagen, Informatik-Report 313

[1] http://dna.fernuni-hagen.de/secondo/.

Güting RH, Schneider M (2005) Moving objects databases. Academic Press, USA. ISBN 978-0-12-088799-6

Kim DH, Ryo KH, Kim HS (2000) A spatiotemporal database model and query language. J Syst Softw 55:129–149

Pelekis N, Theodoulidis B, Kopanakis I, Theodoridis Y (2004) Literature review of spatio-temporal database models. Knowl Eng Rev 19:235–274. doi:10.1017/S026988890400013X

Tomkiewicz SM, Fuller MR, Kie JG, Bates KK (2010) Global positioning system and associated technologies in animal behaviour and ecological research. Philos Trans R Soc B 365:2163–2176. doi:10.1098/rstb.2010.0090

Urbano F, Cagnacci F, Calenge C, Dettki H, Cameron A, Neteler M (2010) Wildlife tracking data management: a new vision. Philos Trans R Soc B 365:2177–2185. doi:10.1098/rstb%202010.0081

Chapter 2
Storing Tracking Data in an Advanced Database Platform (PostgreSQL)

Ferdinando Urbano and Holger Dettki

Abstract The state-of-the-art technical tool for effectively and efficiently managing tracking data is the spatial relational database. Using databases to manage tracking data implies a considerable effort for those who are not already familiar with these tools, but this is necessary to be able to deal with the data coming from the new sensors. Moreover, the time spent to learn databases will be largely paid back with the time saved for the management and processing of the data. In this chapter, you are guided through how to set up a new database in which you will create a table to accommodate the test GPS data sets. You create a new table in a dedicated schema. We describe how to upload the raw GPS data coming from five sensors deployed on roe deer in the Italian Alps into the database and provide further insight into time-related database data types and into the creation of additional database users. The reference software platform used is the open source PostgreSQL with its spatial extension PostGIS. This tool is introduced with its graphical interface pgAdmin. All the examples provided (SQL code) and technical solutions proposed are tuned on this software, although most of the code can be easily adapted for other platforms. The book is focused on the applications of spatial databases to the specific domain of movement ecology: to properly understand the content of this guide and to replicate the proposed database structure and tools, you will need a general familiarity with GIS, wildlife tracking and database programming.

Keywords PostgreSQL · GPS tracking · Data management · Spatial database

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

H. Dettki
Umeå Center for Wireless Remote Animal Monitoring (UC-WRAM),
Department of Wildlife, Fish, and Environmental Studies,
Swedish University of Agricultural Sciences (SLU), Skogsmarksgränd,
SE-901 83 Umeå, Sweden
e-mail: holger.dettki@slu.se

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_2,

Introduction

The first step in the data management process is the acquisition of data recorded by GPS sensors deployed on animals. This can be a complex process that can also change depending on the GPS sensor provider and exceeds the scope of this book. The procedure presented in this chapter assumes that the raw information recorded by the GPS sensors is already available in the form of plain text (*.txt) or comma delimited (*.csv) files. In the Special Topic below, we give some insight on how this can be accomplished.

Special Topic: **Data acquisition from GPS sensors**

Depending on the GPS sensor provider and the technical solutions for data telemetry present in the GPS sensor units deployed on the animals, the data can be obtained in different ways. For near real-time applications it is important to automate the acquisition process as much as possible to fully exploit the information that the continuous data flow potentially provides and to avoid delay or even data loss. Other applications may be satisfied by eventual downloads, which can be handled manually, as long as not too many sensors are involved over a longer period. Most GPS sensor providers offer a so-called 'Push'[1] solution in combination with proprietary communication software to parse the incoming data from an encrypted format into a text format. For this, some providers allow the users to set up their own receiving stations to receive, e.g. data-SMS or data through a mobile data link, using a local GSM- or UMTS-modem together with the proprietary software to decode and export the data into a text file, while other providers use a company-based receiving system and forward the data by simply sending emails with the data in the email body or as email attachment to the user. The same routine is often chosen by providers when the original raw data are received through satellite systems (e.g. ARGOS, Iridium, Globalstar). Many of these communication software tools can be configured easily to automatically receive data in regular intervals and export it as text files into a pre-defined directory, without any user intervention. The procedure is slightly more complicated when emails are used. However, using simple programming languages like Python or Visual-Basic, any IT programmer can quickly write a program to extract data from an email body or extract an email attachment containing the data. There are also a number of free programs on the Internet which can accomplish this task. One of the most elegant ways to push data from the provider's database into the tracking database is to enable a direct link between the two databases and use triggers to move or copy new data. Although this is rarely a technical problem, unfortunately it often fails due to security considerations on the provider or user side. Other tracking applications do not need near real-time access to the data and may therefore handle the data download manually in regular or irregular intervals. The classical situation is the use of 'store-on-board' GPS units, where all data are stored in internal memory on-board the GPS devices until they are removed from the animal after a pre-defined period. The user then downloads the stored data using a cable connection between the GPS unit and the providers' proprietary communication software to parse and export a text file for each unit. This is called a typical 'Pull'[2] solution, which is offered by nearly every GPS unit provider. When the GPS sensor units are equipped with short-range radio signal telemetry transmitters (UHF/VHF), the user can pull the data manually from

[1] The information is 'pushed' from the unit or the provider to the user without user intervention.

[2] The user 'Pulls' the information manually from the unit or the providers' database without provider intervention.

the units using a custom receiver in the field after approaching the animals. Again, the providers' proprietary communication software can then be used to manually download, parse and export a text file for each unit from the custom receiver. When the GPS units are equipped for automatic data transfer via telemetry into the provider's database, some providers offer a manual download option from their database through, e.g. the Telnet protocol, or from their website. Here usually a manual login is required, after which data for some or all animals can be downloaded as a text file. While any of the 'Pull' options are fine as long as few GPS units are deployed for a relatively short time period, it is advisable to try in any tracking application to use 'Push' services and automation from the beginning, as building solutions around manual 'Pull' services can become very costly in human resources when the amount of deployed units increases over time.

As discussed in the previous chapter, the state-of-the-art technical tool to effectively and efficiently manage tracking data is the spatial relational database (Urbano et al. 2010). In this chapter's exercise, you will be guided through how to set up a new database in which you will create a table to accommodate the test GPS data sets. You will see how to upload data from source files using specific database tools.

The reference software platform used in this guide is PostgreSQL, along with its spatial extension PostGIS. All the examples provided and the technical solutions proposed are tuned on this software, although most of the code can be easily adapted for other platforms. There are many reasons that support the choice of PostgreSQL/PostGIS:

- both are free and open source, so any available financial resources can be used for customisation instead of software licences, and they can be used by research groups working with limited funds;
- PostgreSQL is an advanced and widely used database system and offers many features useful for animal movement data management;
- PostGIS is currently one of the most, if not the most, advanced database spatial extensions available and its development by the IT community is very fast;
- PostGIS includes support for raster data, a dedicated geography spatial data type, topology and networks, and has a very large library of spatial functions;
- there is a wide, active and very collaborative community for both PostgreSQL and PostGIS;
- there is very good documentation for both PostgreSQL and PostGIS (manuals, tutorials, books, wiki, blogs);
- PostgreSQL and PostGIS widely implement standards, which make them highly interoperable with a large set of other tools for data management, analysis, visualisation and dissemination;
- they are available for all the most common operating systems and CPU architectures, notably x86 and x86_64.

Although we recommend using PostgreSQL/PostGIS to develop your data management system, a valid open source alternative is SpatiaLite[3] if the number of sensors and researchers involved is small and basic functionality is needed.

[3] http://ww.gaia-gis.it/spatialite/.

SpatiaLite has a more limited set of functions compared with PostgreSQL/PostGIS and is not designed for concurrent access for multiple users, but it has the advantage of being a single and portable file with no need for any software installation. Another option is to use existing e-infrastructures dedicated to wildlife tracking data, such as WRAM[4] or Movebank[5]. These have the advantage of offering a Web-based 'ready to use' management system, with a lot of additional features for data sharing. Data are hosted in an external server managed by these projects, so no database maintenance is needed. Movebank offers also a large set of tools to support collaborations among scientists with both local and global perspectives. On the other hand, these systems are designed for large numbers of users and have limited support for specific customisation. It is also easy to foresee that in the future data management will be increasingly provided by GPS sensor vendors as a basic service.

As mentioned earlier, technical explanations about the basics of GIS, database programming and wildlife tracking are not covered in these exercises. When needed, references to specific documentation (e.g. Web pages, books, articles, tutorials) are given. We recommend that readers become familiar with PostgreSQL and PostGIS before reading this text. In particular, whenever you need any information on these two software tools, we suggest you begin by consulting the official documentation[6].

This book is focused on the applications of these technical tools to the specific domain of movement ecology. We also encourage you to use the database management interface pgAdmin[7] to build and manage PostgreSQL databases. It is installed automatically with PostgreSQL. It has a user-friendly graphical interface and a set of tools to facilitate interaction with the database. On the pgAdmin website, you can find the necessary documentation. In addition, the Web-based tool phpPgAdmin[8] can be used to manage a PostgreSQL database and retrieve data remotely without installing any software on the client side.

Create a New Database

The first step to create the database is the installation of PostgreSQL. Once you have downloaded[9] and installed[10] the software (the release used to test the code in this guide is 9.2), you can use the 'Application Stack Builder' (included with

[4] http://www.slu.se/wram/.

[5] http://www.movebank.org/.

[6] http://www.postgresql.org/docs/, http://postgis.refractions.net/documentation/.

[7] http://www.pgadmin.org/.

[8] http://phppgadmin.sourceforge.net/.

[9] http://www.postgresql.org/download/.

[10] http://wiki.postgresql.org/wiki/Detailed_installation_guide/.

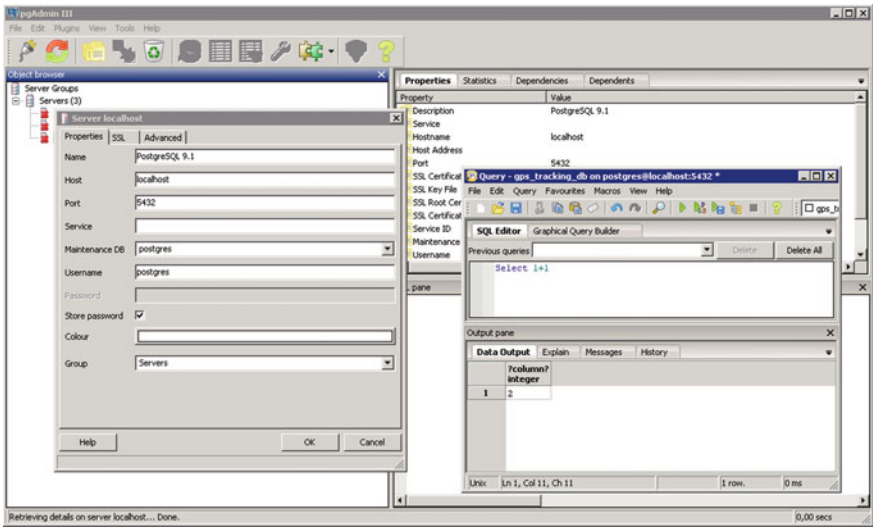

Fig. 2.1 pgAdmin GUI to PostgreSQL. On the *left* is the interface to introduce the connection parameters, and on the *right* is the SQL editor window

PostgreSQL) to get other useful tools, in particular the spatial extension PostGIS (the release used as reference here is 2.0).

During the installation process, you will be able to create the database superuser (the default 'postgres' will be used throughout this guide; don't forget the password!) and set the value for the port (default '5432'). You will need this information together with the IP address of your computer to connect to the database. In case you work directly on the computer where the database is installed, the IP address is also aliased as 'localhost'. If you want your database to be remotely accessible, you must verify that the port is open for external connections. The exercises in this guide use a test data set that includes information from five GPS Vectronic Aerospace GmbH sensors deployed on five roe deer monitored in the Italian Alps (Monte Bondone, Trento), kindly provided by Fondazione Edmund Mach, Trento, Italy. This information is complemented with a variety of (free) environmental layers (locations of meteorological stations, road networks, administrative units, land cover, DEM and NDVI time series). The test data set is available at http://extras.springer.com.

Once you have your PostgreSQL system up and running, you can start using it with the help of SQL queries. Note that you can run SQL code from a PSQL command-line interface[11] or from a graphical SQL interface. The PostgreSQL graphical user interface pgAdmin makes it possible to create all the database objects with user-friendly tools that drive users to define all the required information. Figure 2.1 shows an example of the pgAdmin graphical interface.

[11] http://www.postgresql.org/docs/9.2/static/app-psql.html.

The very first thing to do, even before importing the raw sensor data, is to create a new database with the SQL code[12]

```
CREATE DATABASE gps_tracking_db
  ENCODING = 'UTF8'
  TEMPLATE = template0
  LC_COLLATE = 'C'
  LC_CTYPE = 'C';
```

You could create the database using just the first line of the code[13]. The other lines are added just to be sure that the database will use UTF8 as encoding system and will not be based on any local setting regarding, e.g. alphabets, sorting, or number formatting. This is very important when you work in an international environment where different languages (and therefore characters) can potentially be used.

Create a New Table and Import Raw GPS Data

Now you connect to the database (in pgAdmin you have to double-click on the icon of your database) in order to create a schema (a kind of 'folder' where you store a set of information[14]). By default, a database comes with the 'public' schema; it is good practice, however, to use different schemas to store user data. Here, you create a new schema called *main*:

```
CREATE SCHEMA main;
```

And then you can add a comment to describe the schema:

```
COMMENT ON SCHEMA main IS 'Schema that stores all the GPS tracking core
data.';
```

Comments are stored into the database. They are not strictly needed, but adding a description to every object that you create is a good practice and an important element of effective documentation for your system.

Before importing the GPS data sets into the database, it is recommended that you examine the source data (usually .dbf, .csv, or .txt files) with a spreadsheet or a text editor to see what information is contained. Every GPS brand/model can produce

[12] You can also find a plain text file with the SQL code proposed in the book in the trackingDB_code.zip file available at http://extras.springer.com.

[13] http://www.postgresql.org/docs/9.2/static/sql-createdatabase.html.

[14] http://www.postgresql.org/docs/9.2/static/ddl-schemas.html.

different information, or at least organise this information in a different way, as unfortunately no consolidated standards exist yet (see Chap. 13). The idea is to import raw data (as they are when received from the sensors) into the database and then process them to transform data into information. Once you identify which attributes are stored in the original files, you can create the structure of a table with the same columns, with the correct data types. The list of the main data types available in PostgreSQL/PostGIS is available in the official PostgreSQL documentation[15]. You can find the GPS data sets in .csv files included in the trackingDB_datasets.zip file with test data in the sub-folder \tracking_db\data\sensors_data.

The SQL code that generates the same table structure of the source files within the database, which is called here *main.gps_data*[16], is

```
CREATE TABLE main.gps_data(
  gps_data_id serial,
  gps_sensors_code character varying,
  line_no integer,
  utc_date date,
  utc_time time without time zone,
  lmt_date date,
  lmt_time time without time zone,
  ecef_x integer,
  ecef_y integer,
  ecef_z integer,
  latitude double precision,
  longitude double precision,
  height double precision,
  dop double precision,
  nav character varying(2),
  validated character varying(3),
  sats_used integer,
  ch01_sat_id integer,
  ch01_sat_cnr integer,
  ch02_sat_id integer,
  ch02_sat_cnr integer,
  ch03_sat_id integer,
  ch03_sat_cnr integer,
  ch04_sat_id integer,
  ch04_sat_cnr integer,
  ch05_sat_id integer,
  ch05_sat_cnr integer,
  ch06_sat_id integer,
  ch06_sat_cnr integer,
  ch07_sat_id integer,
  ch07_sat_cnr integer,
  ch08_sat_id integer,
  ch08_sat_cnr integer,
```

[15] http://www.postgresql.org/docs/9.2/static/datatype.html.

[16] 'Main' is the name of the schema where the table will be created, while 'gps_data' is the name of the table. Any object in the database is referred by combining the name of the schema and the name of the object (e.g. table) separated by a dot ('.').

```
  ch09_sat_id integer,
  ch09_sat_cnr integer,
  ch10_sat_id integer,
  ch10_sat_cnr integer,
  ch11_sat_id integer,
  ch11_sat_cnr integer,
  ch12_sat_id integer,
  ch12_sat_cnr integer,
  main_vol double precision,
  bu_vol double precision,
  temp double precision,
  easting integer,
  northing integer,
  remarks character varying
);
COMMENT ON TABLE main.gps_data
IS 'Table that stores raw data as they come from the sensors (plus the ID of
the sensor).';
```

In a relational database, each table should have a primary key: a field (or combination of fields) that uniquely identifies each record. In this case, you added a serial[17] field (*gps_data_id*) not present in the original file. As a serial data type, it is managed by the database and will be unique for each record. You set this field as the primary key of the table:

```
ALTER TABLE main.gps_data
  ADD CONSTRAINT gps_data_pkey
  PRIMARY KEY(gps_data_id);
```

To keep track of database changes, it is useful to add another field to store the time when each record was inserted in the table. The default for this field can be automatically set using the current time using the function *now*()[18]:

```
ALTER TABLE main.gps_data
  ADD COLUMN insert_timestamp timestamp with time zone
  DEFAULT now();
```

If you want to prevent the same record from being imported twice, you can add a unique constraint on the combination of the fields *gps_sensors_code* and *line_no*:

```
ALTER TABLE main.gps_data
  ADD CONSTRAINT unique_gps_data_record
  UNIQUE(gps_sensors_code, line_no);
```

[17] http://www.postgresql.org/docs/9.2/static/datatype-numeric.html#DATATYPE-SERIAL.

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

In case a duplicated record is imported, the whole import procedure fails. You must verify if the above condition on the two fields is reasonable in your case (e.g. the GPS sensor might produce two records with the same values for GPS sensor code and line number). This check also implies some additional time in the import stage.

You are now ready to import the GPS data sets. There are many ways to do it. The main one is to use the *COPY*[19] (*FROM*) command setting the appropriate parameters (*COPY* plus the name of the target table, with the list of column names in the same order as they are in the source file, then *FROM* with the path to the file and *WITH* followed by additional parameters; in this case, 'csv' specifies the format of the source file, *HEADER* means that the first line in the source file is a header and not a record and *DELIMITER* ';' defines the field separator in the source file). Do not forget to change the *FROM* argument to match the actual location of the file on your computer:

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01438.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';')
```

If PostgreSQL complains that date is out of range, check the standard date format used by your database:

```
SHOW datestyle;
```

If it is not 'ISO, DMY' (Day, Month, Year), then you have to set the date format in the same session of the *COPY* statement:

```
SET SESSION datestyle = "ISO, DMY";
```

If the original files are in .dbf format, you can use the pgAdmin tool 'Shapefile and .dbf importer'. In this case, you do not have to create the structure of the table before importing because it is done automatically by the tool, that tries to guess the

[19] See http://www.postgresql.org/docs/9.2/static/sql-copy.html or http://wiki.postgresql.org/wiki/COPY. If you want to upload on a server a file that is located in another machine, you have to use the command '\COPY'; see http://www.postgresql.org/docs/9.2/static/app-psql.html for more information. pgAdmin offers a user-friendly interface for data upload from text files.

right data type for each attribute. This might save some time but you lose control over the definition of data types (e.g. time can be stored as a text value).

Let us also import three other (sensor GSM02927 will be imported in Chap. 4) GPS data sets using the same *COPY* command:

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01508.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01511.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01512.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

Special Topic: **Time and date data type in PostgreSQL**

The management of time and date is more complicated that it may seem. Often, time and date are recorded and stored by the sensors as two separate elements, but the information

that identifies the moment when the GPS position is registered is made of the combination of the two (a so-called 'timestamp'). Moreover, when you have a timestamp, it always refers to a specific time zone. The same moment has a different local time according to the place where you are located. If you do not specify the correct time zone, the database assumes that it is the same as the database setting (usually derived from the local computer setting). This can potentially generate ambiguities and errors. PostgreSQL offers a lot of tools to manage time and date[20]. We strongly suggest using the data type *timestamp with time zone* instead of the simpler but more prone to errors *timestamp without time zone*.

To determine the time zone set in your database you can run

```
SHOW time zone;
```

You can run the following SQL codes to explore how the database manages different specifications of the time and date types:

```
SELECT
  '2012-09-01'::DATE AS date1,
  '12:30:29'::TIME AS time1,
  ('2012-09-01' || ' ' || '12:30:29') AS timetext;
```

In the result below, the data type returned by PostgreSQL are respectively *date*, *time without time zone* and *text*.

```
   date1     |   time1    |       timetext
-------------+------------+---------------------
 2012-09-01 | 12:30:29 | 2012-09-01 12:30:29
```

Here you have some examples of how to create a timestamp data type in PostgreSQL:

```
SELECT
  '2012-09-01'::DATE + '12:30:29'::TIME AS timestamp1,
  ('2012-09-01' || ' ' || '12:30:29')::TIMESTAMP WITHOUT TIME ZONE AS
  timestamp2,
  '2012-09-01 12:30:29+00'::TIMESTAMP WITH TIME ZONE AS timestamp3;
```

In this case, the data type of the first two fields returned is *timestamp without time zone*, while the third one is *timestamp with time zone* (the output can vary according to the default time zone of your database server):

```
     timestamp1       |      timestamp2       |        timestamp3
----------------------+-----------------------+------------------------
 2012-09-01 12:30:29 | 2012-09-01 12:30:29 | 2012-09-01 12:30:29+00
```

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

You can see what happens when you specify the time zone and when you ask for the *timestamp with time zone* from a *timestamp without time zone* (the result will depend on the default time zone of your database server):

```
SELECT
  '2012-09-01 12:30:29 +0'::TIMESTAMP WITH TIME ZONE AS timestamp1,
  ('2012-09-01'::DATE + '12:30:29'::TIME) AT TIME ZONE 'utc' AS timestamp2,
  ('2012-09-01 12:30:29'::TIMESTAMP WITHOUT TIME ZONE)::TIMESTAMP WITH TIME
  ZONE AS timestamp3;
```

The result for a server located in Italy (time zone +02 in summer time) is

```
       timestamp1        |       timestamp2        |       timestamp3
-------------------------+-------------------------+-------------------------
 2012-09-01 14:30:29+02 | 2012-09-01 14:30:29+02 | 2012-09-01 12:30:29+02
```

You can easily extract part of the timestamp, including *epoch* (number of seconds from 1st January 1970, a format that in some cases can be convenient as it expresses a timestamp as an integer):

```
SELECT
  EXTRACT (MONTH FROM '2012-09-01 12:30:29 +0'::TIMESTAMP WITH TIME ZONE) AS
  month1,
  EXTRACT (EPOCH FROM '2012-09-01 12:30:29 +0'::TIMESTAMP WITH TIME ZONE) AS
  epoch1;
```

The expected result is

```
 month1 |   epoch1
--------+------------
      9 | 1346502629
```

In this last example, you set a specific time zone (EST—Eastern Standard Time, which has an offset of −5 h compared to UTC[21]) for the current session:

```
SET timezone TO 'EST';
SELECT now() AS time_in_EST_zone;
```

Here you do the same using UTC as reference zone:

```
SET timezone TO 'UTC';
SELECT now() AS time_in_UTC_zone;
```

[21] Coordinated Universal Time (UTC) is the primary time standard by which the world regulates clocks and time. For most purposes, UTC is synonymous with GMT, but GMT is no longer precisely defined by the scientific community.

You can compare the results of the two queries to see the difference. To permanently change the reference time zone to UTC, you have to edit the file postgresql.conf[22]. In most of the applications related to movement ecology, this is probably the best option as GPS uses this reference.

Finalise the Database: Defining GPS Acquisition Timestamps, Indexes and Permissions

In the original GPS data file, no timestamp field is present. Although the table *main.gps_data* is designed to store data as they come from the sensors, it is convenient to have an additional field where date and time are combined and where the correct time zone is set (in this case UTC). To do so, you first add a field of *timestamp with time zone* type. Then, you fill it (with an *UPDATE* statement) from the time and date fields. In a later exercise, you will see how to automatise this step using triggers.

```
ALTER TABLE main.gps_data
  ADD COLUMN acquisition_time timestamp with time zone;
UPDATE main.gps_data
  SET acquisition_time = (utc_date + utc_time) AT TIME ZONE 'UTC';
```

Now, the table is ready. Next you can add some indexes[23], which are data structures that improve the speed of data retrieval operations on a database table at the cost of slower writes and the use of more storage space. Database indexes work in a similar way to a book's table of contents: you have to add an extra page and update it whenever new content is added, but then searching for specific sections will be much faster. You have to decide on which fields you create indexes for by considering what kind of query will be performed most often in the database. Here, you add indexes on the *acquisition_time* and the *sensor_id* fields, which are probably two key attributes in the retrieval of data from this table:

```
CREATE INDEX acquisition_time_index
  ON main.gps_data
  USING btree (acquisition_time );
CREATE INDEX gps_sensors_code_index
  ON main.gps_data
  USING btree (gps_sensors_code);
```

[22] http://www.postgresql.org/docs/9.2/static/config-setting.html.

[23] http://www.postgresql.org/docs/9.2/static/sql-createindex.html.

As a simple example, you can now retrieve data using specific selection criteria (GPS positions in May from the sensor GSM01438). Let us retrieve data from the collar 'GSM01512' during the month of May (whatever the year), and order them by their acquisition time:

```
SELECT
  gps_data_id AS id, gps_sensors_code AS sensor_id, latitude, longitude,
  acquisition_time
FROM
  main.gps_data
WHERE
  gps_sensors_code = 'GSM01512' and EXTRACT(MONTH FROM acquisition_time) = 5
ORDER BY
  acquisition_time
LIMIT 10;
```

The first records (*LIMIT 10* returns just the first 10 records; you can remove this condition to have the full list of records) of the result of this query are

```
  id   | sensor_id | latitude | longitude |    acquisition_time
-------+-----------+----------+-----------+------------------------
 11906 | GSM01512  | 46.00563 |  11.05291 | 2006-05-01 00:01:01+00
 11907 | GSM01512  | 46.00630 |  11.05352 | 2006-05-01 04:02:54+00
 11908 | GSM01512  | 46.00652 |  11.05326 | 2006-05-01 08:01:03+00
 11909 | GSM01512  | 46.00437 |  11.05536 | 2006-05-01 12:02:40+00
 11910 | GSM01512  | 46.00720 |  11.05297 | 2006-05-01 16:01:23+00
 11911 | GSM01512  | 46.00709 |  11.05339 | 2006-05-01 20:00:53+00
 11912 | GSM01512  | 46.00723 |  11.05346 | 2006-05-02 00:00:54+00
 11913 | GSM01512  | 46.00649 |  11.05251 | 2006-05-02 04:01:54+00
 11914 | GSM01512  |          |           | 2006-05-02 08:03:06+00
 11915 | GSM01512  | 46.00687 |  11.05386 | 2006-05-02 12:01:24+00
```

One of the main advantages of an advanced database management system like PostgreSQL is that the database can be accessed by a number of users at the same time, keeping the data always in a single version with a proper management of concurrency. This ensures that the database maintains the ACID (atomicity, consistency, isolation, durability) principles in an efficient manner. Users can be different, with different permissions. Most commonly, you have a single administrator that can change the database, and a set of users that can just read the data. As an example, you create here a user[24] (login *basic_user*, password *basic_user*) and grant *read* permission for the *main.gps_data* table and all the objects that will be created in the *main* schema in the future:

[24] http://www.postgresql.org/docs/9.2/static/sql-createrole.html.

```
CREATE ROLE basic_user LOGIN
  PASSWORD 'basic_user'
  NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE NOREPLICATION;
GRANT SELECT ON ALL TABLES
  IN SCHEMA main
  TO basic_user;
ALTER DEFAULT PRIVILEGES
  IN SCHEMA main
  GRANT SELECT ON TABLES
  TO basic_user;
```

Permissions can also be associated with user groups, in which case new users can be added to each group and will inherit all the related permissions on database objects. Setting a permission policy in a complex multi-user environment requires an appropriate definition of data access at different levels and it is out of the scope of this guide. You can find more information on the official PostgreSQL documentation[25].

Export Data and Backup

There are different ways to export a table or the results of a query to an external file. One is to use the command *COPY* (*TO*)[26]. An example is

```
COPY (
  SELECT gps_data_id, gps_sensors_code, latitude, longitude, acquisition_time,
  insert_timestamp
  FROM main.gps_data)
TO
   'C:\tracking_db\test\export_test1.csv'
   WITH (FORMAT csv, HEADER, DELIMITER ';');
```

Another possibility is to use the pgAdmin interface: in the SQL console select 'Query'/'Execute to file'. Other database interfaces have similar tools.

A proper backup policy for a database is important to securing all your valuable data and the information that you have derived through data processing. In general, it is recommended to have frequent (scheduled) backups (e.g. once a day) for schemas that change often and less frequent backups (e.g. once a week) for schemas (if any) that occupy a larger disk size and do not change often. In case of

[25] http://www.postgresql.org/docs/9.2/static/user-manag.html.

[26] http://www.postgresql.org/docs/9.2/static/sql-copy.html, http://wiki.postgresql.org/wiki/COPY.

ad hoc updates of the database, you can run specific backups. PostgreSQL offers very good tools for database backup and recovery[27]. The two main tools to backup are as follows:

- pg_dump.exe: extracts a PostgreSQL database or part of the database into a script file or other archive file (pg_restore.exe is used to restore the database);
- pg_dumpall.exe: extracts a PostgreSQL database cluster (i.e. all the databases created inside the same installation of PostgreSQL) into a script file (e.g. including database setting, roles).

These are not SQL commands but executable commands that must run from a command-line interpreter (with Windows, the default command-line interpreter is the program cmd.exe, also called Command Prompt). pgAdmin also offers a graphic interface for backing up and restoring the database. Moreover, it is also important to keep a copy of the original raw data files, particularly those generated by sensors.

Reference

Urbano F, Cagnacci F, Calenge C, Dettki H, Cameron A, Neteler M (2010) Wildlife tracking data management: a new vision. Philos Trans R Soc B 365:2177–2185. doi:10.1098/rstb.2010.0081

[27] http://www.postgresql.org/docs/9.2/static/backup.html.

Chapter 3
Extending the Database Data Model: Animals and Sensors

Ferdinando Urbano

Abstract GPS positions are used to describe animal movements and to derive a large set of information, for example, about animals' behaviour, social interactions and environmental preferences. GPS data are related to (and must be integrated with) many other sources of information that together can be used to describe the complexity of movement ecology. This can be achieved through proper database data modelling, which depends on a clear definition of the biological context of a study. Particularly, data modelling becomes a key step when database systems manage many connected data sets that grow in size and complexity: it permits easy updates of the database structure to accommodate the changing goals, constraints and spatial scales of studies. In this chapter's exercise, you will extend your database (see Chap. 2) with two new tables to integrate ancillary information useful to interpreting GPS data: one for GPS sensors and the other for animals.

Keywords Data modelling · GPS tracking · Data management · Spatial database

Introduction

GPS positions are used to describe animal movements and to derive a large set of information, for example, on animals' behaviour, social interactions and environmental preferences. GPS data are related to (and must be integrated with) many other information that together can be used to describe the complexity of movement ecology. This can be achieved through proper database data modelling. A data model describes what types of data are stored and how they are organised.

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

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_3,

It can be seen as the conceptual representation of the real world in the database structures that include data objects (i.e. tables) and their mutual relationships. In particular, data modelling becomes a key step when database systems grow in size and complexity, and user requirements become more sophisticated: it permits easy updates of the database structure to accommodate the changing goals, constraints, and spatial scales of studies and the evolution of wildlife tracking systems. Without a rigorous data modelling approach, an information system might lose the flexibility to manage data efficiently in the long term, reducing its utility to a simple storage device for raw data, and thus failing to address many of the necessary requirements.

To model data properly, you have to clearly state the biological context of your study. A logical way to proceed is to define (1) very basic questions on the sample unit, i.e. individual animals and (2) basic questions about data collection.

1. Typically, individuals are the sampling units of an ecological study based on wildlife tracking. Therefore, the first question to be asked while modelling the data is: 'What basic biological information is needed to characterise individuals as part of a sample?' Species, sex and age (or age class) at capture are the main factors which are relevant in all studies. Age classes typically depend on the species[1]. Other information used to characterise individuals could be specific to a study, for example in a study on spatial behaviour of translocated animals, 'resident' or 'translocated' is an essential piece of information linked to individual animals. All these elements should be described in specific tables.
2. A single individual becomes a 'studied unit' when it is fitted with a sensor, in this case to collect position information. First of all, GPS sensors should be described by a dedicated table containing the technical characteristics of each device (e.g. vendor, model). Capture time, or 'device-fitting' time, together with the time and a description of the end of the deployment (e.g. drop-off of the tag, death of the animal), are also essential to the model. The link between the sensors and the animals should be described in a table that states unequivocally when the data collected from a sensor 'become' (and cease to be) bio-logged data, i.e. the period during which they refer to an individual's behaviour. The start of the deployment usually coincides with the moment of capture, but it is not the same thing. Indeed, moment of capture can be the 'end' of one relationship between a sensor and an animal (i.e. when a device is taken off an animal) and at the same time the 'beginning' of another (i.e. another device is fitted instead).

[1] Age class of an animal is not constant for all the GPS positions. The correct age class at any given moment can be derived from the age class at capture and by defining rules that specify when the individual changes from one class to another (for roe deer, you might assume that on 1st April of every year each individual that was a fawn becomes a yearling, and each yearling becomes an adult).

Thanks to the tables 'animals', 'sensors', and 'sensors to animals', and the relationships built among them, GPS data can be linked unequivocally to individuals, i.e. the sampling units.

Some information related to animals can change over time. Therefore, they must be marked with the reference time that they refer to. Examples of typical parameters assessed at capture are age and positivity of association to a disease. Translocation may also coincide with the capture/release time. If this information changes over time according to well-defined rules (e.g. transition from age classes), their value can be dynamically calculated in the database at different moments in time (e.g. using database functions). You will see an example of a function to calculate age class from the information on the age class at capture and the acquisition time of GPS positions for roe deer in Chap. 9.

The basic structure 'animals', 'sensors', 'sensors to animals', and, of course, 'position data', can be extended to take into account the specific goals of each project, the complexity of the real-world problems faced, the technical environment and the available data. Examples of data that can be integrated are capture methodology, handling procedure, use of tranquilizers and so forth, that should be described in a 'captures' table linked to the specific individual (in the table 'animals'). Finally, data referring to individuals may come from several sources, e.g. several sensors or visual observations. In all these cases, the link between data and sample units (individuals) should also be clearly stated by appropriate relationships.

At the moment, there is a single table in the test database that represents raw data from GPS sensors. Now, you can start including more information in new tables to represent other important elements involved in wildlife tracking. This process will continue throughout all the following chapters.

In this chapter's exercise, you will include two new tables: one for GPS sensors and one for animals, with some ancillary tables (age classes, species).

Import Information on GPS Sensors and Add Constraints to the Table

In the subfolder \tracking_db\data\animals and \tracking_db\data\sensors of the test data set[2], you will find two files: animals.csv and gps_sensors.csv. Let us start with data on GPS sensors. First, you have to create a table in the database with the same attributes as the .csv file and then import the data into it. Here is the code of the table structure:

[2] The file with the test data set trackingDB_datasets.zip is part of the Extra Material of the book available at http://extras.springer.com.

```
CREATE TABLE main.gps_sensors(
  gps_sensors_id integer,
  gps_sensors_code character varying NOT NULL,
  purchase_date date,
  frequency double precision,
  vendor character varying,
  model character varying,
  sim character varying,
  CONSTRAINT gps_sensors_pkey
    PRIMARY KEY (gps_sensors_id ),
  CONSTRAINT gps_sensor_code_unique
    UNIQUE (gps_sensors_code)
);
COMMENT ON TABLE main.gps_sensors
IS 'GPS sensors catalog.';
```

The only field that is not present in the original file is *gps_sensors_id*. This is an integer[3] used as primary key. You could also use *gps_sensors_code* as primary key, but in many practical situations it is handy to use an integer field.

You add a field to keep track of the timestamp of record insertion:

```
ALTER TABLE main.gps_sensors
  ADD COLUMN insert_timestamp timestamp with time zone DEFAULT now();
```

Now, you can import data using the *COPY* command:

```
COPY main.gps_sensors(
  gps_sensors_id, gps_sensors_code, purchase_date, frequency, vendor, model,
  sim)
FROM
  'C:\tracking_db\data\sensors\gps_sensors.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

At this stage, you have defined the list of GPS sensors that exist in your database. To be sure that you will never have GPS data that come from a GPS sensor that does not exist in the database, you apply a foreign key[4] between *main.gps_data* and *main.gps_sensors*. Foreign keys physically translate the concept of relations among tables.

[3] In some cases, a good recommendation is to use a 'serial' number as primary key to let the database generate a unique code (integer) every time that a new record is inserted. In this exercise, we use an integer data type because the values of the *gps_sensors_id* field are defined in order to be correctly referenced in the exercises of the next chapters.

[4] http://www.postgresql.org/docs/9.2/static/tutorial-fk.html.

```
ALTER TABLE main.gps_data
  ADD CONSTRAINT gps_data_gps_sensors_fkey
  FOREIGN KEY (gps_sensors_code)
  REFERENCES main.gps_sensors (gps_sensors_code)
  MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION;
```

This setting says that in order to delete a record in *main.gps_sensors*, you first have to delete all the associated records in *main.gps_data*. From now on, before importing GPS data from a sensor, you have to create the sensor's record in the *main.gps_sensors* table.

You can add other kinds of constraints to control the consistency of your database. As an example, you check that the date of purchase is after 2000-01-01. If this condition is not met, the database will refuse to insert (or modify) the record and will return an error message.

```
ALTER TABLE main.gps_sensors
  ADD CONSTRAINT purchase_date_check
  CHECK (purchase_date > '2000-01-01'::date);
```

Import Information on Animals and Add Constraints to the Table

Now, you repeat the same process for data on animals. Analysing the animals' source file (animals.csv), you can derive the fields of the new *main.animals* table:

```
CREATE TABLE main.animals(
  animals_id integer,
  animals_code character varying(20) NOT NULL,
  name character varying(40),
  sex character(1),
  age_class_code integer,
  species_code integer,
  note character varying,
  CONSTRAINT animals_pkey PRIMARY KEY (animals_id)
);
COMMENT ON TABLE main.animals
IS 'Animals catalog with the main information on individuals.';
```

As for *main.gps_sensors*, in your operational database, you can use the *serial* data type for the *animals_id* field. Age class (at capture) and species are attributes that can only have defined values. To enforce consistency in the database, in these cases, you can use lookup tables. Lookup tables store the list and the description of all possible values referenced by specific fields in different tables and constitute the

definition of the valid domain. It is recommended to keep them in a separated schema to give the database a more readable and clear data structure. Therefore, you create a *lu_tables* schema:

```
CREATE SCHEMA lu_tables;
  GRANT USAGE ON SCHEMA lu_tables TO basic_user;
COMMENT ON SCHEMA lu_tables
IS 'Schema that stores look up tables.';
```

You set as default that the user *basic_user* will be able to run *SELECT* queries on all the tables that will be created in this schema:

```
ALTER DEFAULT PRIVILEGES
  IN SCHEMA lu_tables
  GRANT SELECT ON TABLES
  TO basic_user;
```

Now, you create a lookup table for species:

```
CREATE TABLE lu_tables.lu_species(
  species_code integer,
  species_description character varying,
  CONSTRAINT lu_species_pkey
  PRIMARY KEY (species_code)
);
COMMENT ON TABLE lu_tables.lu_species
IS 'Look up table for species.';
```

You populate it with some values (just roe deer code will be used in our test data set):

```
INSERT INTO lu_tables.lu_species
  VALUES (1, 'roe deer');
INSERT INTO lu_tables.lu_species
  VALUES (2, 'rein deer');
INSERT INTO lu_tables.lu_species
  VALUES (3, 'moose');
```

You can do the same for age classes:

```
CREATE TABLE lu_tables.lu_age_class(
  age_class_code integer,
  age_class_description character varying,
  CONSTRAINT lage_class_pkey
  PRIMARY KEY (age_class_code)
);
COMMENT ON TABLE lu_tables.lu_age_class
IS 'Look up table for age classes.';
```

You populate it with some values[5]:

```
INSERT INTO lu_tables.lu_age_class
  VALUES (1, 'fawn');
INSERT INTO lu_tables.lu_age_class
  VALUES (2, 'yearling');
INSERT INTO lu_tables.lu_age_class
  VALUES (3, 'adult');
```

At this stage, you can create the foreign keys between the *main.animals* table and the two lookup tables:

```
ALTER TABLE main.animals
  ADD CONSTRAINT animals_lu_species
  FOREIGN KEY (species_code)
  REFERENCES lu_tables.lu_species (species_code)
  MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE main.animals
  ADD CONSTRAINT animals_lu_age_class
  FOREIGN KEY (age_class_code)
  REFERENCES lu_tables.lu_age_class (age_class_code)
  MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION;
```

For sex class of deer, you do not expect to have more than the two possible values: female and male (stored in the database as ‘f’ and ‘m’ to simplify data input). In this case, instead of a lookup table you can set a check on the field:

```
ALTER TABLE main.animals
  ADD CONSTRAINT sex_check
  CHECK (sex = 'm' OR sex = 'f');
```

Whether it is better to use a lookup table or a check must be evaluated case by case, mainly according to the number of admitted values and the possibility that you will want to add new values in the future.

You should also add a field to keep track of the timestamp of record insertion:

```
ALTER TABLE main.animals
  ADD COLUMN insert_timestamp timestamp with time zone DEFAULT now();
```

[5] These categories are based on roe deer; other species might need a different approach.

As a last step, you import the values from the file:

```
COPY main.animals(
  animals_id,animals_code, name, sex, age_class_code, species_code)
FROM
  'C:\tracking_db\data\animals\animals.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

To test the result, you can retrieve the animals' data with the extended species and age class description:

```
SELECT
  animals.animals_id AS id,
  animals.animals_code AS code,
  animals.name,
  animals.sex,
  lu_age_class.age_class_description AS age_class,
  lu_species.species_description AS species
FROM
  lu_tables.lu_age_class,
  lu_tables.lu_species,
  main.animals
WHERE
  lu_age_class.age_class_code = animals.age_class_code
  AND
  lu_species.species_code = animals.species_code;
```

The result of the query is:

```
 id | code |    name    | sex | age_class | species
----+------+------------+-----+-----------+----------
  1 | F09  | Daniela    | f   | adult     | roe deer
  2 | M03  | Agostino   | m   | adult     | roe deer
  3 | M06  | Sandro     | m   | adult     | roe deer
  4 | F10  | Alessandra | f   | adult     | roe deer
  5 | M10  | Decimo     | m   | adult     | roe deer
```

You can also create this query with the pgAdmin tool 'Graphical Query Builder' (Fig. 3.1).

First Elements of the Database Data Model

In Fig. 3.2, you have a schematic representation of the tables created so far in the database, and their relationships. As you can see, the table *animals* is linked with foreign keys to two tables in the schema where the lookup tables are stored. In fact, the tables *lu_species* and *lu_age_class* contain the admitted values for the related fields in the *animals* table. The table *gps_data*, which contains the raw data

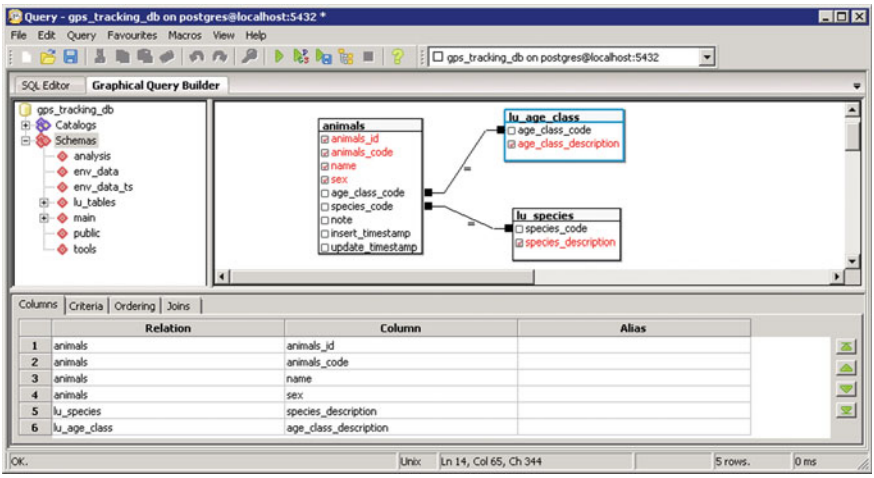

Fig. 3.1 pgAdmin GUI interface to create queries

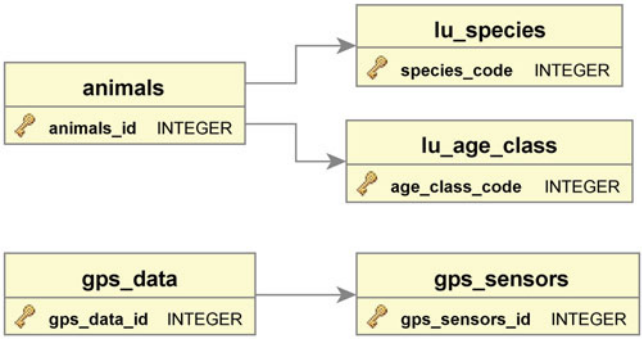

Fig. 3.2 Tables stored in the database at the end of the exercise for this chapter. The arrows identify links between tables connected by foreign keys

coming from GPS sensors, is linked to the table *gps_sensors*, where all the existing GPS sensors are stored with a set of ancillary information. A foreign key creates a dependency, for example, the table *gps_data* cannot store data from a GPS sensor if the sensor is not included in the *gps_sensors* table.

Note that, at this point, there is no relation between animals and the GPS data. In other words, it is impossible to retrieve positions of a given animal, but only of a given collar. Moreover, you cannot distinguish between GPS positions recorded when the sensors were deployed on the animals and those that were recorded for example, in the researcher's office before the deployment. You will see in the following chapter how to associate GPS positions with animals.

Chapter 4
From Data to Information: Associating GPS Positions with Animals

Ferdinando Urbano

Abstract When position data are received from GPS sensors, they are not explicitly associated with any animal. Linking GPS data to animals is a key step in the data management process. This can be achieved using the information on the deployments of GPS sensors on animals (when sensors started and ceased to be deployed on the animals). In the case of a continuous data flow, the transformation of GPS positions into animal locations must be automated in order to have GPS data imported and processed in real-time. In the exercise for this chapter, you extend the database built in Chaps. 2 and 3 with two new tables, *gps_sensors_animals* and *gps_data_animals*, and a set of dedicated database triggers and functions that add the necessary tools to properly manage the association of GPS positions with animals.

Keywords Automated data flow · Triggers · Real-time data processing

Introduction

When position data are received from GPS sensors, they are not explicitly associated with any animal.[1] Linking GPS data to animals is a key step in the data management process. This can be achieved using the information about the

[1] Sometimes raw data from GPS sensors (e.g. .txt or .csv files produced by the software that manages data acquisition on the client side) have no explicit information on the sensor code itself inside the file. In these cases, the sensor code can be usually derived from the data source (e.g. from the name of the file). In this exercise, it is assumed that the sensor code is already in the raw data file.

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

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_4,

deployments of GPS sensors (tags) on animals (when sensors started and ceased to be deployed on the animals). If the acquisition time of a GPS position is recorded after the start and before the end of the deployment of the sensor on an animal, the position is associated with that animal. This process not only links the GPS sensor to the animal that is wearing it, but also excludes the positions recorded when the sensor was not deployed on the animal. In the case of a large number of sensors and animals, this process cannot be managed manually, but requires some dedicated, and possibly automated, tools. Moreover, the process of associating GPS positions and animals must be able to manage changes in the information about sensor deployment. For example, hours or even days can pass before the death of an animal tagged with a GPS sensor is discovered. During this time, the GPS positions acquired in near real-time are associated with the animal. This is an error, as the positions recorded between the death and its detection by researchers are not valid and must be 'disassociated' from the animal. This approach also efficiently manages the redeployment of a GPS sensor recovered from an animal (because of, e.g. the end of battery or death of the animal) to another animal, and the deployment of a new GPS sensor on an animal previously monitored with another GPS sensor.

The key point is to properly store the information on the deployment of sensors on animals in a dedicated table in the database, taking into consideration that each animal can be monitored with multiple GPS sensors (most likely at different times) and each sensor can be reused on multiple animals (no more than one at a time). This corresponds to a many-to-many relationship between animals and GPS sensors, where the main attribute is the time range (the start and end of deployment[2]). Making reference to the case of GPS (but with a general validity), this information can be stored in a *gps_sensors_animals* table where the ID of the sensor, the ID of the animal and the start and end timestamps of deployment are stored.

A possible approach to store the records that are associated with animals is to create a new table, which could be called *gps_data_animals*, where a list of derived fields can be eventually added to the basic animals ID, GPS sensors ID, acquisition time and coordinates. Figure 4.1 illustrates a general picture of this database data model structure. This new table duplicates part of the information stored in the original *gps_data* table, and the two tables must be kept synchronised. On the other hand, there are many advantages of this structure over the alternative approach with a single table (*gps_data*) where all the original data from GPS sensors (including GPS positions not associated with animals) are stored together with other information (e.g. the animal ID, environmental attributes, movement parameters):

- *gps_data* cannot be easily synchronised with the data source if too many additional fields (i.e. calculated after data are imported into the database) are present in the table;

[2] When the sensor is still deployed on the animal, the end of deployment can be set to null.

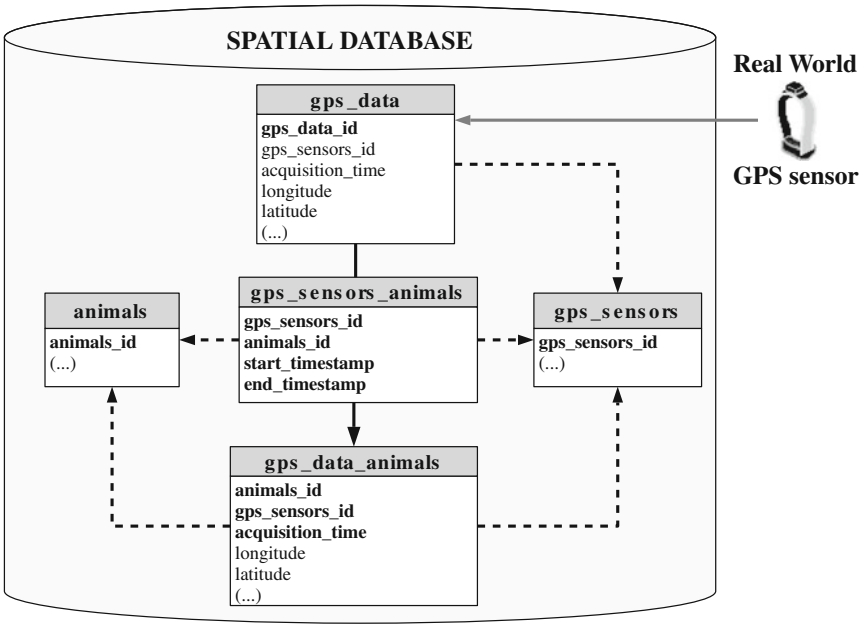

Fig. 4.1 Core database data model structure. Devices are associated with animals for defined time ranges in the table 'gps_sensors_animals', which depends on the tables 'animals' and 'gps_sensors' (dependencies are represented by dashed arrows). Once GPS position data are uploaded into the database in the 'gps_data' table, they are assigned to a specific animal (i.e. the animal wearing the sensor at the acquisition time of the GPS position) using the information from the table 'gps_sensors_animals'. Thus, a new table ('gps_data_animals') is filled (solid arrow) with the identifier of the individual, its position, the acquisition time and the identifier of the GPS device. The names of the fields that uniquely identify a record are marked in bold

- if sensors from different vendors or different models of the same vendor are used, you might have file formats with a different set of fields; in this case, it is complicated to merge all the information from each source in a single *gps_data* table;
- in a table containing just those GPS positions associated with animals, performance improves because of the reduced number of records and fields (the fields not relevant for analysis, e.g. the list of satellites can be kept only in the *gps_data* table);
- with the additional *gps_data_animals* table, it is easier and more efficient to manage a system of tags to mark potential outliers and to share and disseminate the relevant information (you would lose the information on outliers if *gps_data* is synchronised with the original data set, i.e. the text file from the sensors).

In this book, we use the table *gps_data* as an exact copy of raw data as they come from GPS sensors, while *gps_data_animals* is used to store and process the

information that is used to monitor and study animals' movements. In a way, *gps_data* is a 'system' table used as an intermediate step for the data import process.

Storing Information on GPS Sensor Deployments on Animals

The first step to associating GPS positions with animals is to create a table to accommodate information on the deployment of GPS sensors on animals:

```
CREATE TABLE main.gps_sensors_animals(
  gps_sensors_animals_id serial NOT NULL,
  animals_id integer NOT NULL,
  gps_sensors_id integer NOT NULL,
  start_time timestamp with time zone NOT NULL,
  end_time timestamp with time zone,
  notes character varying,
  insert_timestamp timestamp with time zone DEFAULT now(),
  CONSTRAINT gps_sensors_animals_pkey
    PRIMARY KEY (gps_sensors_animals_id ),
  CONSTRAINT gps_sensors_animals_animals_id_fkey
    FOREIGN KEY (animals_id)
    REFERENCES main.animals (animals_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT gps_sensors_animals_gps_sensors_id_fkey
    FOREIGN KEY (gps_sensors_id)
    REFERENCES main.gps_sensors (gps_sensors_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT time_interval_check
    CHECK (end_time > start_time)
);
COMMENT ON TABLE main.gps_sensors_animals
IS 'Table that stores information of deployments of sensors on animals.';
```

Now, your table is ready to be populated. The general way of populating this kind of table is the manual entry of the information. In our case, you can use the test data set stored in the .csv file included in the test data set[3] \tracking_db\data\sensors_animals\gps_sensors_animals.csv:

```
COPY main.gps_sensors_animals(
  animals_id, gps_sensors_id, start_time, end_time, notes)
FROM
  'c:\tracking_db\data\sensors_animals\gps_sensors_animals.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

[3] The file with the test data set trackingDB_datasets.zip is part of the Extra Material of the book available at http://extras.springer.com.

Once the values in this table are updated, you can use an SQL statement to obtain the id of the animal related to each GPS position. Here, an example of a query to retrieve the codes of the animal and GPS sensor, the acquisition time and the coordinates (to make the query easier to read, aliases are used for the name of the tables and of the attributes):

```
SELECT
  deployment.gps_sensors_id AS sensor,
  deployment.animals_id AS animal,
  data.acquisition_time,
  data.longitude::numeric(7,5) AS long,
  data.latitude::numeric(7,5) AS lat
FROM
  main.gps_sensors_animals AS deployment,
  main.gps_data AS data,
  main.gps_sensors AS gps
WHERE
  data.gps_sensors_code = gps.gps_sensors_code AND
  gps.gps_sensors_id = deployment.gps_sensors_id AND
  (
    (data.acquisition_time >= deployment.start_time AND
     data.acquisition_time <= deployment.end_time)
    OR
    (data.acquisition_time >= deployment.start_time AND
     deployment.end_time IS NULL)
  )
ORDER BY
  animals_id, acquisition_time
LIMIT 10;
```

In the query, three tables are involved: *main.gps_sensors_animals*, *main.gps_data*, and *main.gps_sensors*. This is because in the *main.gps_data*, where raw data from the sensors are stored, the *gps_sensors_id* is not present, and thus, the table *main.gps_sensors* is necessary to convert the *gps_sensors_code* into the corresponding *gps_sensors_id.* You can see that in the *WHERE* part of the statement two cases are considered: when the acquisition time is after the start and before the end of the deployment, and when the acquisition time is after the start of the deployment and the end is *NULL* (which means that the sensor is still deployed on the animal). The first 10 records returned by this *SELECT* statement are

```
 sensor | animal |    acquisition_time    |   long   |   lat
--------+--------+------------------------+----------+----------
      4 |      1 | 2005-10-18 20:00:54+00 | 11.04413 | 46.01096
      4 |      1 | 2005-10-19 00:01:23+00 | 11.04454 | 46.01178
      4 |      1 | 2005-10-19 04:02:22+00 | 11.04515 | 46.00793
      4 |      1 | 2005-10-19 08:03:08+00 | 11.04567 | 46.00600
      4 |      1 | 2005-10-20 20:00:53+00 | 11.04286 | 46.01015
      4 |      1 | 2005-10-21 00:00:48+00 | 11.04172 | 46.01051
      4 |      1 | 2005-10-21 04:00:53+00 | 11.04089 | 46.01028
      4 |      1 | 2005-10-21 08:01:42+00 | 11.04429 | 46.00669
      4 |      1 | 2005-10-21 12:03:11+00 |          |
      4 |      1 | 2005-10-21 16:01:16+00 | 11.04622 | 46.00684
```

From GPS Positions to Animal Locations

Now, you can permanently store this information in a new table (*main.gps_data_animals*) that is used as reference for data analysis, visualisation and dissemination. Below is the SQL code that generates this table:

```
CREATE TABLE main.gps_data_animals(
  gps_data_animals_id serial NOT NULL,
  gps_sensors_id integer,
  animals_id integer,
  acquisition_time timestamp with time zone,
  longitude double precision,
  latitude double precision,
  insert_timestamp timestamp with time zone DEFAULT now(),
  CONSTRAINT gps_data_animals_pkey
    PRIMARY KEY (gps_data_animals_id),
  CONSTRAINT gps_data_animals_animals_fkey
    FOREIGN KEY (animals_id)
    REFERENCES main.animals (animals_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT gps_data_animals_gps_sensors
    FOREIGN KEY (gps_sensors_id)
    REFERENCES main.gps_sensors (gps_sensors_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION
);
COMMENT ON TABLE main.gps_data_animals
IS 'GPS sensors data associated with animals wearing the sensor.';
CREATE INDEX gps_data_animals_acquisition_time_index
  ON main.gps_data_animals
  USING BTREE (acquisition_time);
CREATE INDEX gps_data_animals_animals_id_index
  ON main.gps_data_animals
  USING BTREE (animals_id);
```

At this point, you can feed this new table with the data in the table *gps_data* and use *gps_sensors_animals* to derive the id of the animals:

```
INSERT INTO main.gps_data_animals (
  animals_id, gps_sensors_id, acquisition_time, longitude, latitude)
SELECT
  gps_sensors_animals.animals_id,
  gps_sensors_animals.gps_sensors_id,
  gps_data.acquisition_time, gps_data.longitude,
  gps_data.latitude
FROM
  main.gps_sensors_animals, main.gps_data, main.gps_sensors
WHERE
  gps_data.gps_sensors_code = gps_sensors.gps_sensors_code AND
  gps_sensors.gps_sensors_id = gps_sensors_animals.gps_sensors_id AND
  (
    (gps_data.acquisition_time>=gps_sensors_animals.start_time AND
     gps_data.acquisition_time<=gps_sensors_animals.end_time)
    OR
    (gps_data.acquisition_time>=gps_sensors_animals.start_time AND
     gps_sensors_animals.end_time IS NULL)
  );
```

Another possibility is to simultaneously create and populate the table *main.gps_data_animals* by using '*CREATE TABLE main.gps_data_animals AS*' instead of '*INSERT INTO main.gps_data_animals*' in the previous query and then adding the primary and foreign keys and indexes to the table.

Timestamping Changes in the Database Using Triggers

In the case of a continuous data flow, it is useful to automatise this step in order to have GPS data imported in real time into *gps_data_animals*. To achieve these results, you can use two powerful tools: functions and triggers. It might be convenient to store all functions and ancillary tools in a defined schema:

```
CREATE SCHEMA tools
  AUTHORIZATION postgres;
  GRANT USAGE ON SCHEMA tools TO basic_user;
COMMENT ON SCHEMA tools
IS 'Schema that hosts all the functions and ancillary tools used for the
database.';
ALTER DEFAULT PRIVILEGES
  IN SCHEMA tools
  GRANT SELECT ON TABLES
  TO basic_user;
```

Special Topic: **PostgreSQL functions and triggers**

A function[4] is a program code that is implemented inside the database using SQL or a set of other languages (e.g. PSQL, Python, C). Functions allow you to create complex processes and algorithms when plain SQL queries alone cannot do the job. Once created, a function becomes part of the database library and can be called inside SQL queries. Here is a simple example of an SQL function that makes the sum of two input integers:

```
CREATE FUNCTION tools.test_add(integer, integer)
  RETURNS integer AS 'SELECT $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

The variables *$1* and *$2* are the first and second input parameters. You can test it with

[4] For those who are interested in creating their own functions, many resources are available on the Internet, e.g.:
http://www.postgresql.org/docs/9.2/static/sql-createfunction.html
http://www.postgresql.org/docs/9.2/static/xfunc-sql.html.

```
SELECT tools.test_add(2,7);
```

The result is '9'.

In the framework of this guide, you do not necessarily need to create your own functions, but you must be aware of the possibility offered by these tools and be able to understand and use existing functions. Advanced users can adapt them according to their specific needs.

A trigger[5] is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed on a particular table in the database. The trigger fires a specific function to perform some actions *BEFORE* or *AFTER* records are *DELETED*, *UPDATED*, or *INSERTED* in a table. The trigger function must be defined before the trigger itself is created. The trigger function must be declared as a function taking no arguments and returning type trigger. For example, when you insert a new record in a table, you can modify the values of the attributes before they are uploaded or you can update another table that should be affected by this new upload.

As a first simple example of a trigger, you add a field to the table *gps_data_animals* where you register the timestamp of the last modification (update) of each record in order to keep track of the changes in the table. This field can have *now()* as default value (current time) when data are inserted the first time:

```
ALTER TABLE main.gps_data_animals
  ADD COLUMN update_timestamp timestamp with time zone DEFAULT now();
```

Once you have created the field, you need a function called by a trigger to update this field whenever a record is updated. The SQL to generate the function is

```
CREATE OR REPLACE FUNCTION tools.timestamp_last_update()
RETURNS trigger AS
$BODY$BEGIN
IF NEW IS DISTINCT FROM OLD THEN
  NEW.update_timestamp = now();
END IF;
RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.timestamp_last_update()
IS 'When a record is updated, the update_timestamp is set to the current
time.';
```

Here is the code for the trigger that calls the function:

```
CREATE TRIGGER update_timestamp
  BEFORE UPDATE
  ON main.gps_data_animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.timestamp_last_update();
```

[5] http://www.postgresql.org/docs/9.2/static/triggers.html.

You have to initialise the existing records in the table, as the trigger/function was not yet created when data were uploaded:

```
UPDATE main.gps_data_animals
  SET update_timestamp = now();
```

You can now repeat the same operation for the other tables in the database:

```
ALTER TABLE main.gps_sensors
  ADD COLUMN update_timestamp timestamp with time zone DEFAULT now();
ALTER TABLE main.animals
  ADD COLUMN update_timestamp timestamp with time zone DEFAULT now();
ALTER TABLE main.gps_sensors_animals
  ADD COLUMN update_timestamp timestamp with time zone DEFAULT now();
CREATE TRIGGER update_timestamp
  BEFORE UPDATE
  ON main.gps_sensors
  FOR EACH ROW
  EXECUTE PROCEDURE tools.timestamp_last_update();
CREATE TRIGGER update_timestamp
  BEFORE UPDATE
  ON main.gps_sensors_animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.timestamp_last_update();
CREATE TRIGGER update_timestamp
  BEFORE UPDATE
  ON main.animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.timestamp_last_update();
UPDATE main.gps_sensors
  SET update_timestamp = now();
UPDATE main.gps_sensors_animals
  SET update_timestamp = now();
UPDATE main.animals
  SET update_timestamp = now();
```

Another interesting application of triggers is the automation of the *acquisition_time* computation when a new record is inserted into the *gps_data* table:

```
CREATE OR REPLACE FUNCTION tools.acquisition_time_update()
RETURNS trigger AS
$BODY$BEGIN
  NEW.acquisition_time = ((NEW.utc_date + NEW.utc_time) at time zone 'UTC');
  RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.acquisition_time_update()
IS 'When a record is inserted, the acquisition_time is composed from
utc_date and utc_time.';
```

```
CREATE TRIGGER update_acquisition_time
  BEFORE INSERT
  ON main.gps_data
  FOR EACH ROW
  EXECUTE PROCEDURE tools.acquisition_time_update();
```

Automation of the GPS Data Association with Animals

With triggers and functions, you can automatise the upload from *gps_data* to *gps_data_animals* of records that are associated with animals (for sensors deployed on animals). First, you have to create the function that will be called by the trigger:

```
CREATE OR REPLACE FUNCTION tools.gps_data2gps_data_animals()
RETURNS trigger AS
$BODY$ begin
INSERT INTO main.gps_data_animals (
  animals_id, gps_sensors_id, acquisition_time, longitude, latitude)
SELECT
  gps_sensors_animals.animals_id, gps_sensors_animals.gps_sensors_id,
  NEW.acquisition_time, NEW.longitude, NEW.latitude
FROM
  main.gps_sensors_animals, main.gps_sensors
WHERE
  NEW.gps_sensors_code = gps_sensors.gps_sensors_code AND
  gps_sensors.gps_sensors_id = gps_sensors_animals.gps_sensors_id AND
  (
    (NEW.acquisition_time >= gps_sensors_animals.start_time AND
     NEW.acquisition_time <= gps_sensors_animals.end_time)
    OR
    (NEW.acquisition_time >= gps_sensors_animals.start_time AND
     gps_sensors_animals.end_time IS NULL)
  );
RETURN NULL;
END
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.gps_data2gps_data_animals()
IS 'Automatic upload data from gps_data to gps_data_animals.';
```

Then, you create a trigger that calls the function whenever a new record is uploaded into *gps_data*:

```
CREATE TRIGGER trigger_gps_data_upload
  AFTER INSERT
  ON main.gps_data
  FOR EACH ROW
  EXECUTE PROCEDURE tools.gps_data2gps_data_animals();
COMMENT ON TRIGGER trigger_gps_data_upload ON main.gps_data
IS 'Upload data from gps_data to gps_data_animals whenever a new record is
inserted.';
```

You can test this function by adding the last GPS sensor not yet imported:

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM02927.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

Data are automatically processed and imported into the table *gps_data_animals* including the correct association with the animal wearing the sensor.

Consistency Checks on the Deployments Information

The management of the association between animals and GPS sensors can be improved using additional, more sophisticated tools. A first example is the implementation of consistency checks on the *gps_sensors_animals* table. As this is a key table, it is important to avoid illogical associations. The two most evident constraints are that the same sensor cannot be worn by two animals at the same time and that no more than one GPS sensor can be deployed on the same animal at the same time (this assumption can be questionable in case of other sensors, but in general can be considered valid for GPS). To prevent any impossible overlaps in animal/sensor deployments, you have to create a trigger on both insertion and updates of records in *gps_animals_sensors* that verifies the correctness of the new values. In case of invalid values, the insert/modify statement is aborted and an error message is raised by the database. Here is an example of code for this function:

```
CREATE OR REPLACE FUNCTION tools.gps_sensors_animals_consistency_check()
RETURNS trigger AS
$BODY$
DECLARE
  deletex integer;
BEGIN
SELECT
  gps_sensors_animals_id
INTO
  deletex
FROM
  main.gps_sensors_animals b
WHERE
  (NEW.animals_id = b.animals_id OR NEW.gps_sensors_id = b.gps_sensors_id)
  AND
  (
  (NEW.start_time > b.start_time AND NEW.start_time < b.end_time)
  OR
  (NEW.start_time > b.start_time AND b.end_time IS NULL)
  OR
  (NEW.end_time > b.start_time AND NEW.end_time < b.end_time)
  OR
  (NEW.start_time < b.start_time AND NEW.end_time > b.end_time)
  OR
  (NEW.start_time < b.start_time AND NEW.end_time IS NULL )
  OR
  (NEW.end_time > b.start_time AND b.end_time IS NULL)
);
IF deletex IS not NULL THEN
  IF TG_OP = 'INSERT' THEN
    RAISE EXCEPTION 'This row is not inserted: Animal-sensor association not
    valid: (the same animal would wear two different GPS sensors at the same
    time or the same GPS sensor would be deployed on two animals at the same
    time).';
    RETURN NULL;
  END IF;
  IF TG_OP = 'UPDATE' THEN
    IF deletex != OLD.gps_sensors_animals_id THEN
      RAISE EXCEPTION 'This row is not updated: Animal-sensor association
      not valid (the same animal would wear two different GPS sensors at the same
      time or the same GPS sensor would be deployed on two animals at the same
      time).';
      RETURN NULL;
    END IF;
  END IF;
END IF;
RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.gps_sensors_animals_consistency_check()
IS 'Check if a modified or insert row in gps_sensors_animals is valid (no
impossible time range overlaps of deployments).';
```

Here is an example of the trigger to call the function:

```
CREATE TRIGGER gps_sensors_animals_changes_consistency
  BEFORE INSERT OR UPDATE
  ON main.gps_sensors_animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.gps_sensors_animals_consistency_check();
```

You can test this process by trying to insert a deployment of a GPS sensor in the *gps_sensors_animals* table in a time interval that overlaps the association of the same sensor on another animal:

```
INSERT INTO main.gps_sensors_animals
  (animals_id, gps_sensors_id, start_time, end_time, notes)
VALUES
  (2,2,'2004-10-23 20:00:53 +0','2005-11-28 13:00:00 +0','Ovelapping
   sensor');
```

You should receive an error message like:

```
********** Error **********
ERROR: This row is not inserted: Animal-sensor association not valid: (the same
animal would wear two different GPS sensors at the same time or the same GPS
sensor would be deployed on two animals at the same time).
SQL state: P0001
```

Synchronisation of gps_sensors_animals and gps_data_animals

In an operational environment where data are managed in (near) real-time, it happens that the information about the association between animals and sensors changes over time. A typical example is the death of an animal: this event is usually discovered with a delay of some days. In the meantime, GPS positions are received and associated with the animals in the *gps_data_animals* table. When the new information on the deployment time range is registered in *gps_sensors_animals*, the table *gps_data_animals* must be changed accordingly. It is highly desirable that any change in the table *gps_sensors_animals* is automatically reflected in *gps_data_animals*. It is possible to use triggers to keep the two tables automatically synchronised. Below you have an example of a trigger function to implement this procedure. The code manages the three possible operations: delete, insert and modification of records in the *gps_sensors_animals* table. For each case, it checks whether GPS positions previously associated with an animal are no longer valid (and if so, deletes them from the table *gps_data_animals*) and whether GPS positions previously not associated with the animal should now be linked (and if so, adds them to the table *gps_data_animals*). The function is complex because the process is complex. You can use as it is, or go through it if you need to adapt.

```
CREATE OR REPLACE FUNCTION tools.gps_sensors_animals2gps_data_animals()
RETURNS trigger AS
$BODY$ BEGIN

IF TG_OP = 'DELETE' THEN
  DELETE FROM
    main.gps_data_animals
  WHERE
    animals_id = OLD.animals_id AND
    gps_sensors_id = OLD.gps_sensors_id AND
    acquisition_time >= OLD.start_time AND
    (acquisition_time <= OLD.end_time OR OLD.end_time IS NULL);
  RETURN NULL;
END IF;

IF TG_OP = 'INSERT' THEN
  INSERT INTO
    main.gps_data_animals (gps_sensors_id, animals_id, acquisition_time,
    longitude, latitude)
  SELECT
    NEW.gps_sensors_id, NEW.animals_id, gps_data.acquisition_time,
    gps_data.longitude, gps_data.latitude
  FROM
    main.gps_data, main.gps_sensors
  WHERE
    NEW.gps_sensors_id = gps_sensors.gps_sensors_id AND
    gps_data.gps_sensors_code = gps_sensors.gps_sensors_code AND
    gps_data.acquisition_time >= NEW.start_time AND
    (gps_data.acquisition_time <= NEW.end_time OR NEW.end_time IS NULL);
  RETURN NULL;
END IF;

IF TG_OP = 'UPDATE' THEN
  DELETE FROM
    main.gps_data_animals
  WHERE
    gps_data_animals_id IN (
      SELECT
        d.gps_data_animals_id
      FROM
        (SELECT
          gps_data_animals_id, gps_sensors_id, animals_id, acquisition_time
        FROM
          main.gps_data_animals
        WHERE
          gps_sensors_id = OLD.gps_sensors_id AND
          animals_id = OLD.animals_id AND
          acquisition_time >= OLD.start_time AND
          (acquisition_time <= OLD.end_time OR OLD.end_time IS NULL)
        ) d
```

```
    LEFT OUTER JOIN
      (SELECT
        gps_data_animals_id, gps_sensors_id, animals_id, acquisition_time
      FROM
        main.gps_data_animals
      WHERE
        gps_sensors_id = NEW.gps_sensors_id AND
        animals_id = NEW.animals_id AND
        acquisition_time >= NEW.start_time AND
        (acquisition_time <= NEW.end_time OR NEW.end_time IS NULL)
      ) e
    ON
      (d.gps_data_animals_id = e.gps_data_animals_id)
    WHERE e.gps_data_animals_id IS NULL);
INSERT INTO
  main.gps_data_animals (gps_sensors_id, animals_id, acquisition_time,
  longitude, latitude)
SELECT
  u.gps_sensors_id, u.animals_id, u.acquisition_time, u.longitude,
  u.latitude
FROM
  (SELECT
    NEW.gps_sensors_id AS gps_sensors_id, NEW.animals_id AS animals_id,
    gps_data.acquisition_time AS acquisition_time, gps_data.longitude AS
    longitude, gps_data.latitude AS latitude
  FROM
    main.gps_data, main.gps_sensors
  WHERE
    NEW.gps_sensors_id = gps_sensors.gps_sensors_id AND
    gps_data.gps_sensors_code = gps_sensors.gps_sensors_code AND
    gps_data.acquisition_time >= NEW.start_time AND
    (acquisition_time <= NEW.end_time OR NEW.end_time IS NULL)
  ) u
LEFT OUTER JOIN
  (SELECT
    gps_data_animals_id, gps_sensors_id, animals_id, acquisition_time
  FROM
    main.gps_data_animals
  WHERE
    gps_sensors_id = OLD.gps_sensors_id AND
    animals_id = OLD.animals_id AND
    acquisition_time >= OLD.start_time AND
    (acquisition_time <= OLD.end_time OR OLD.end_time IS NULL)
  ) w
ON
    (u.gps_sensors_id = w.gps_sensors_id AND
    u.animals_id = w.animals_id AND
    u.acquisition_time = w.acquisition_time )
  WHERE
    w.gps_data_animals_id IS NULL;
  RETURN NULL;
END IF;
```

```
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.gps_sensors_animals2gps_data_animals()
IS 'When a record in gps_sensors_animals is deleted OR updated OR inserted,
this function synchronizes this information with gps_data_animals.';
```

Here is the code of the trigger to call the function:

```
CREATE TRIGGER synchronize_gps_data_animals
  AFTER INSERT OR UPDATE OR DELETE
  ON main.gps_sensors_animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.gps_sensors_animals2gps_data_animals();
```

In Fig. 4.2, you have a complete picture of the part of the database data model related to GPS data.

It is important to emphasise that triggers are powerful tools for automating the data flow. The drawback is that they will slow down the data import process. This note is also valid for indexes, which speed up queries but imply some additional computation during the import stage. In the case of frequent uploads (or modification) of very large data sets at once, the use of the proposed triggers could significantly decrease performance. In these cases, you can more quickly process the data in a later stage after they are imported into the database and are therefore available to users. The best approach must be identified according to the specific goals, constraints and characteristics of your application. In this guide, we use as reference the management of data coming from a set of sensors deployed on animals, transmitting data in near real time, where the import step will include just a few thousand locations at a time.

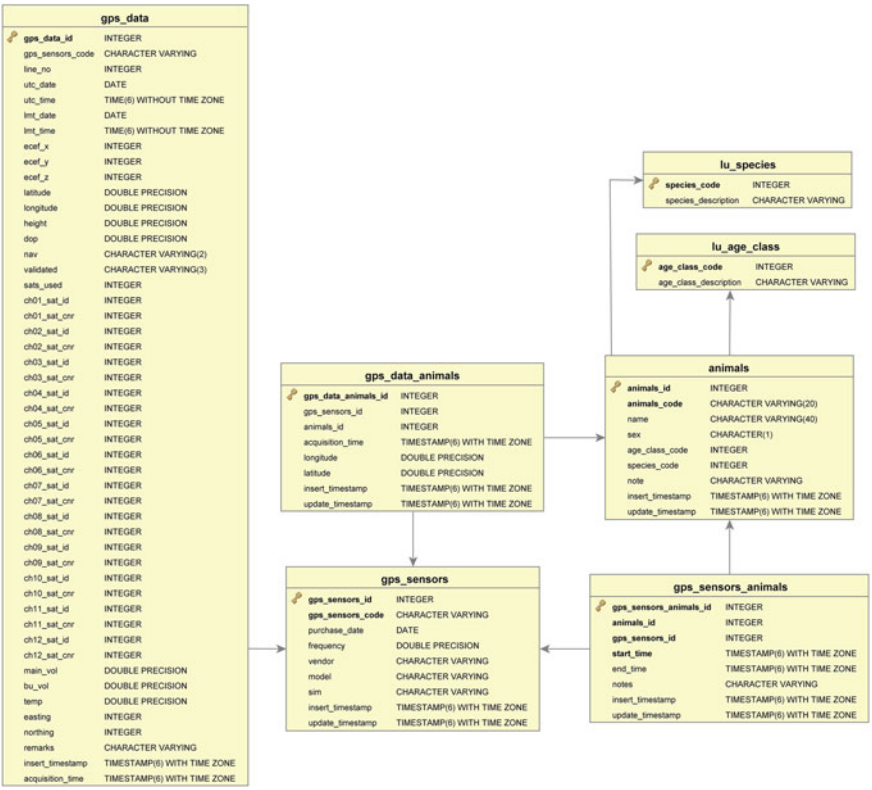

Fig. 4.2 Schema of the database data model related to GPS data

Chapter 5
Spatial is not Special: Managing Tracking Data in a Spatial Database

Ferdinando Urbano and Mathieu Basille

Abstract A wildlife tracking data management system must include the capability to explicitly deal with the spatial properties of movement data. GPS tracking data are sets of spatiotemporal objects (locations), and the spatial component must be properly managed. You will now extend the database built in Chaps. 2, 3 and 4, adding spatial functionalities through the PostgreSQL spatial extension called PostGIS. PostGIS introduces spatial data types (both vector and raster) and a large set of SQL spatial functions and tools, including spatial indexes. This possibility essentially allows you to build a GIS using the capabilities of relational databases. In this chapter, you will start to familiarise yourself with spatial SQL and implement a system that automatically transforms the GPS coordinates generated by GPS sensors from a pair of numbers into spatial objects.

Keywords PostGIS · Spatial data · GPS tracking · Animal movement

Introduction

A wildlife tracking data management system must include the capability to explicitly deal with the spatial component of movement data. GPS tracking data are sets of spatiotemporal objects (locations) that have to be properly managed.

At the moment, your data are stored in the database and the GPS positions are linked to individuals. While time is correctly managed, coordinates are still just

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

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_5,

two decimal numbers (longitude and latitude) and not spatial objects. It is therefore not possible to find the distance between two points, or the length of a trajectory, or the speed and angle of the step between two locations. In this chapter, you will learn how to add a spatial extension to your database and transform the coordinates into a spatial element (i.e. a point).

Until a few years ago, the spatial information produced by GPS sensors was managed and analysed using dedicated software (GIS) in file-based data formats (e.g. shapefiles). Nowadays, the most advanced approaches in data management consider the spatial component of objects (e.g. a moving animal) as one of its many attributes: thus, while understanding the spatial nature of your data is essential for proper analysis, from a software perspective, spatial is (increasingly) not special. Spatial databases are the technical tools needed to implement this perspective. They integrate spatial data types (vector and raster) together with standard data types that store the objects' other (non-spatial) associated attributes. Spatial data types can be manipulated by SQL through additional commands and functions for the spatial domain. This possibility essentially allows you to build a GIS using the existing capabilities of relational databases. Moreover, while dedicated GIS software is usually focused on analyses and data visualisation, providing a rich set of spatial operations, few are optimised for managing large spatial data sets (in particular, vector data) and complex data structures. Spatial databases, in turn, allow both advanced management and spatial operations that can be efficiently undertaken on a large set of elements. This combination of features is becoming essential, as with animal movement data sets the challenge is now on the extraction of synthetic information from very large data sets rather than on the extrapolation of new information (e.g. kernel home ranges from VHF data) from limited data sets with complex algorithms.

Spatial databases can perform a wide variety of spatial operations, typically

- spatial measurements: calculate the distance between points, polygon area, etc.;
- spatial functions: modify existing features to create new ones, for example, by providing a buffer around them, intersecting features, etc.;
- spatial predicates: allow true/false queries such as 'is there a village located within a kilometre of the area where an animal is moving?';
- constructor functions: create new features specifying the vertices (points of nodes) which can make up lines, and if the first and last vertexes of a line are identical, the feature can also be of the type polygon (a closed line);
- observer functions: query the database to return specific information about a feature such as the location of the centre of a home range.

Spatial databases use spatial indexes[1] to speed up database operations and optimise spatial queries.

Today, practically all major relational databases offer native spatial information capabilities and functions in their products, including PostgreSQL (PostGIS),

[1] http://workshops.opengeo.org/postgis-intro/indexing.html.

IBM DB2 (Spatial Extender), SQL Server (SQL Server 2008 Spatial), Oracle (ORACLE Spatial), Informix (Spatial Datablade), MYSQL (Spatial Extension) and SQLite (Spatialite), while ESRI ArcSDE is a middleware application that can spatially enable different DBMSs.

The Open Geospatial Consortium[2] (OGC) created the Simple Features specification and sets standards for adding spatial functionality to database systems. The spatial database extension that implements the largest number of OGC specifications is the open source tool PostGIS for PostgreSQL, and this is one of the main reasons why PostgreSQL has been chosen as the reference database for this book. A good reference guide[3] for PostGIS can be found in Obe and Hsu (2011) and Corti et al. (2014).

In this chapter, you will extend your database with the spatial dimension of GPS locations and start to familiarise yourself with spatial SQL. You will implement a system that automatically transforms coordinates from a pair of numbers into spatial objects. You are also encouraged to explore the PostGIS documentation where the long list of available tools is described.

Spatially Enable the Database

You can install PostGIS using the Application Stack Builder that comes with the PostgreSQL, or directly from the PostGIS website[4]. Once PostGIS is installed, enable it in your database with the following SQL command:

```
CREATE EXTENSION postgis;
```

Now, you can use and exploit all the features offered by PostGIS in your database. The vector objects (points, lines and polygons) are stored in a specific field of your tables as spatial data types. This field contains the structured list of vertexes, i.e. coordinates of the spatial object, and also includes its reference system. The PostGIS spatial (vectors) data types are not topological, although, if needed, PostGIS has a dedicated topological extension[5]. As you will explore in Chaps. 6 and 7, PostGIS can also manage raster data.

---

[2] http://www.opengeospatial.org/.

[3] There are also many online resources where you can find useful introductions to get started (and become proficient) with PostGIS. Here are some suggestions:
http://postgis.refractions.net/
http://postgis.net/docs/manual-2.0/
http://postgisonline.org/tutorials/
http://trac.osgeo.org/postgis/wiki/UsersWikiTutorials.

[4] http://postgis.net/install.

[5] http://postgis.refractions.net/docs/Topology.html.

An important setting is the reference system used to store (and manage) your GPS position data set. In PostGIS, reference systems are identified with a spatial reference system identifier (SRID) and more specifically the SRID implementation defined by the European Petroleum Survey Group[6] (EPSG). Each reference system is associated with a unique code. GPS coordinates are usually expressed from sensors as longitude/latitude, using the WGS84 geodetic datum (geographic coordinates). This is a reference system that is used globally, using angular coordinates related to an ellipsoid that approximates the earth's shape. As a consequence, it is not correct to apply functions that are designed to work on Euclidean space, because on an ellipsoid, the closest path between two points is not a straight line but an arc. In fact, most of the environmental layers available in a given area are projected in a plane reference system (e.g. Universal Transverse Mercator, UTM).

PostGIS has two main groups of spatial vector data types: *geometry*, which works with any kind of spatial reference, and *geography*, which is specific for geographic coordinates (latitude and longitude WGS84).

Special Topic: **Geometry and geography data type**

The PostGIS *geography* data type[7] provides native support for spatial features represented in 'geographic' coordinates (latitude/longitude WGS84). Geographic coordinates are spherical coordinates expressed in angular units (degrees). Calculations (e.g. areas, distances, lengths, intersections) on the *geometry* data type features are performed using Cartesian mathematics and straight line vectors, while calculations on *geography* data type features are done on the sphere, using more complicated mathematics. For more accurate measurements, the calculations must take the actual spheroidal shape of the world into account, and the mathematics become very complicated. Due to this additional complexity, there are fewer (and slower) functions defined for the *geography* type than for the *geometry* type. Over time, as new algorithms are added, the capabilities of the *geography* type will expand. In any case, it is always possible to convert back and forth between *geometry* and *geography* types.

It is recommended that you not store GPS position data in some projected reference system, but instead keep them as longitude/latitude WGS84. You can later project your features in any other reference system whenever needed. There are two options: they can be stored as *geography* data type or as *geometry* data type, specifying the geographic reference system by its SRID code, which in this case is 4236. The natural choice for geographic coordinates would be the *geography* data type because the *geometry* data type assumes that geographic coordinates refer to Euclidean space. In fact, if you calculate the distance between two points stored as *geometry* data type with SRID 4326, the result will be wrong (latitude and longitude are not planar coordinates so the Euclidean distance between two points makes little sense).

[6] http://www.epsg.org/.

[7] http://postgis.refractions.net/docs/using_postgis_dbmanagement.html#PostGIS_Geography.

At the moment, the *geography* data type is not yet supported by all the PostGIS spatial functions[8]; therefore, it might be convenient to store GPS locations as the *geometry* data type (with the geographic reference system). In this way, you can quickly convert to the *geography* data type for calculation with spherical geometry, or project to any other reference system to use more complex spatial functions and to relate GPS positions with other (projected) environmental data sets (e.g. land cover, digital elevation model, vegetation indexes). Moreover, not all the client applications are able to deal with the *geography* data type. The choice between the *geometry* and *geography* data types also depends on general considerations about performance (*geography* data type involves more precise but also slower computations as it uses a spherical geometry) and data processes to be supported.

Exploring Spatial Functions

Before you create a spatial field for your data, you can explore some very basic tools. First, you can create a point feature:

```
SELECT
  ST_MakePoint(11.001,46.001) AS point;
```

These coordinates are longitude and latitude, although you have not specified (yet) the reference system. The result is

```
                  point
-----------------------------------------------
 01010000008D976E1283002640E3A59BC420004740
```

The long series of characters that are returned depends on how the point is coded in the database. You can easily and transparently see its textual representation (*ST_AsEWKT* or *ST_AsText*):

```
SELECT ST_AsEWKT(
  ST_MakePoint(11.001,46.001)) AS point;
```

In this case, the result is

```
      point
---------------------
 POINT(11.001 46.001)
```

[8] For the list of functions that support the *geography* data type see http://postgis.refractions.net/docs/PostGIS_Special_Functions_Index.html#PostGIS_GeographyFunctions.

You can specify the reference system of your coordinates using *ST_SetSRID*:

```
SELECT ST_AsEWKT(
  ST_SetSRID(ST_MakePoint(11.001,46.001), 4326))AS point;
```

This query returns

```
            point
--------------------------------
SRID=4326;POINT(11.001 46.001)
```

You can project the point in any other reference system. In this example, you project (*ST_Transform*) the coordinates of the point in UTM32 WGS84 (SRID 32632):

```
SELECT
  ST_X(
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.001,46.001), 4326), 32632))::integer
      AS x_utm32,
  ST_y(
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.001,46.001), 4326), 32632))::integer
      AS y_utm32;
```

The result is

```
 x_utm32 | y_utm32
---------+---------
  654938 | 5096105
```

Here, you create a simple function to automatically find the UTM zone at defined coordinates:

```
CREATE OR REPLACE FUNCTION tools.srid_utm(longitude double precision,latitude
  double precision)
RETURNS integer AS
$BODY$
DECLARE
  srid integer;
  lon float;
  lat float;
BEGIN
  lat := latitude;
  lon := longitude;
```

```
IF ((lon > 360 or lon < -360) or (lat > 90 or lat < -90)) THEN
  RAISE EXCEPTION 'Longitude and latitude is not in a valid format (-360 to
  360; -90 to 90)';
ELSEIF (longitude < -180)THEN
  lon := 360 + lon;
ELSEIF (longitude > 180)THEN
  lon := 180 - lon;
END IF;

IF latitude >= 0 THEN
  srid := 32600 + floor((lon+186)/6);
ELSE
  srid := 32700 + floor((lon+186)/6);
END IF;

RETURN srid;
END;
$BODY$
LANGUAGE plpgsql VOLATILE STRICT
COST 100;
COMMENT ON FUNCTION tools.srid_utm(double precision, double precision)
IS 'Function that returns the SRID code of the UTM zone where a point (in
geographic coordinates) is located. For polygons or line, it can be used
giving ST_x(ST_Centroid(the_geom)) and ST_y(ST_Centroid(the_geom)) as
parameters. This function is typically used be used with ST_Transform to
project elements with no prior knowledge of their position.';
```

Here is an example to see the SRID of the UTM zone of the point at coordinates (11.001, 46.001):

```
SELECT TOOLS.SRID_UTM(11.001,46.001) AS utm_zone;
```

The result is

```
 utm_zone
----------
    32632
```

You can use this function to project points when you do not know the UTM zone:

```
SELECT
  ST_AsEWKT(
    ST_Transform(
      ST_SetSRID(ST_MakePoint(31.001,16.001), 4326),
      TOOLS.SRID_UTM(31.001,16.001))
  ) AS projected_point;
```

The result is

```
                  projected_point
-----------------------------------------------------
 SRID=32636;POINT(286087.858226893 1770074.92410008)
```

If you want to allow the user *basic_user* to project spatial data, you have to grant permission on the table *spatial_ref_sys*:

```
GRANT SELECT ON TABLE spatial_ref_sys TO basic_user;
```

Now, you can try to compute the distance between two points. You can try with geographic coordinates as *geometry* data type:

```
SELECT
  ST_Distance(
    ST_SetSRID(ST_MakePoint(11.001,46.001), 4326),
    ST_SetSRID(ST_MakePoint(11.03,46.02), 4326)) AS distance;
```

The result is

```
     distance
--------------------
 0.0346698716467224
```

As you can see, the result is given in the original unit (decimal degrees) because the *geometry* data type, which is the standard setting unless you explicitly specify the *geography* data type, applies the Euclidean distance to the points in geographic coordinates. In fact, distance between coordinates related to a spheroid should not be computed in Euclidean space (the minimum distance is not a straight line but a great circle arc). PostGIS offers many options to get the real distance in meters between two points in geographic coordinates. You can project the points and then compute the distance:

```
SELECT
  ST_Distance(
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.001,46.001), 4326), 32632),
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.03,46.02), 4326),32632)) AS distance;
```

The result (in meters) is

```
    distance
------------------
 3082.64215399684
```

You can also use a specific function to compute distance on a sphere (*ST_Distance_Sphere*):

```
SELECT
  ST_Distance_Sphere(
    ST_SetSRID(ST_MakePoint(11.001,46.001), 4326),
    ST_SetSRID(ST_MakePoint(11.03,46.02), 4326)) AS distance;
```

The result (in meters) is

```
   distance
-----------------
 3078.8604714608
```

A sphere is just a rough approximation of the earth. A better approximation, at cost of more computational time, is given by the function *ST_Distance_Spheroid* where you have to specify the reference ellipsoid:

```
SELECT
  ST_Distance_Spheroid(
    ST_SetSRID(ST_MakePoint(11.001,46.001), 4326),
    ST_SetSRID(ST_MakePoint(11.03,46.02), 4326),
    'SPHEROID["WGS 84",6378137,298.2257223563]') AS distance;
```

The result is

```
    distance
------------------
 3082.95263824183
```

One more option is to 'cast' (transform a data type into another data type using '::') *geometry* as *geography*. Then, you can compute distance and PostGIS will execute this operation taking into account the nature of the reference system:

```
SELECT
  ST_Distance(
    ST_SetSRID(ST_MakePoint(11.001,46.001), 4326)::geography,
    ST_SetSRID(ST_MakePoint(11.03,46.02), 4326)::geography) AS distance;
```

The result is

```
    distance
------------------
 3082.95257067079
```

You can compare the results of the previous queries to see the different outputs. They are all different as a result of the different methods (and associated approximation) used to calculate them. The slowest and most precise is generally thought to be *ST_Distance_Spheroid.*

Another useful feature of PostGIS is the support of 3D spatial objects, which might be relevant, for example, for avian or marine species, or terrestrial species that move in an environment with large altitudinal variations. Here is an example that computes distances in a 2D space using *ST_Distance* and in a 3D space using *ST_3DDistance*, where the vertical displacement is also considered:

```
SELECT
  ST_Distance(
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.001,46.001), 4326), 32632),
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.03,46.02), 4326),32632)) AS distance_2D,
  ST_3DDistance(
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.001,46.001, 0), 4326), 32632),
    ST_Transform(
      ST_SetSRID(ST_MakePoint(11.03,46.02, 1000), 4326),32632)) AS distance_3D;
```

The result is

```
  distance_2d    |   distance_3d
-----------------+-----------------
 3082.64215399684 | 3240.78426458755
```

Not all PostGIS functions support 3D objects, but the number is quickly increasing.

Transforming GPS Coordinates into a Spatial Object

Now, you can create a field with *geometry* data type in your table (2D point feature with longitude/latitude WGS84 as reference system):

```
ALTER TABLE main.gps_data_animals
  ADD COLUMN geom geometry(Point,4326);
```

You can create a spatial index:

```
CREATE INDEX gps_data_animals_geom_gist
  ON main.gps_data_animals
  USING gist (geom );
```

You can now populate it (excluding points that have no latitude/longitude):

```
UPDATE
  main.gps_data_animals
SET
  geom = ST_SetSRID(ST_MakePoint(longitude, latitude),4326)
WHERE
  latitude IS NOT NULL AND longitude IS NOT NULL;
```

At this point, it is important to visualise the spatial content of your tables. PostgreSQL/PostGIS offers no tool for spatial data visualisation, but this can be done by a number of client applications, in particular GIS desktop software like ESRI ArcGIS 10.* or QGIS. QGIS[9] is a powerful and complete open source software. It offers all the functions needed to deal with spatial data. QGIS is the suggested GIS interface because it has many specific tools for managing and visualising PostGIS data. Especially remarkable is the tool 'DB Manager'. In Fig. 5.1, you can see a screenshot of the QGIS interface to insert the connection parameters to the database.

Now, you can use the tool 'Add PostGIS layer' to visualise and explore the GPS position data set (see Fig. 5.2). The example is a view zoomed in on the study area rather than all points, because some outliers (see Chap. 8) are located very far from the main cluster, affecting the default visualisation. In the background, you have OpenStreetMap layer loaded using the 'Openlayer' plugin.

You can also use ArcGIS ESRI 10[10].* to visualise (but not edit, at least at the time of writing this book) your spatial data. Data can be accessed using 'Query layers'[11]. A query layer is a layer or stand-alone table that is defined by an SQL query. Query layers allow both spatial and non-spatial information stored in a (spatial) DBMS to be integrated into GIS projects within ArcMap. When working in ArcMap, you create query layers by defining an SQL query. The query is then run against the tables and viewed in a database, and the result set is added to ArcMap. Query layers behave like any other feature layer or stand-alone table, so they can be used to display data, used as input into a geoprocessing tool or accessed using developer APIs. The query is executed every time the layer is displayed or used in ArcMap. This allows the latest information to be visible

[9] http://www.qgis.org/.

[10] http://www.esri.com/software/arcgis.

[11] http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/Connecting_to_a_database/.

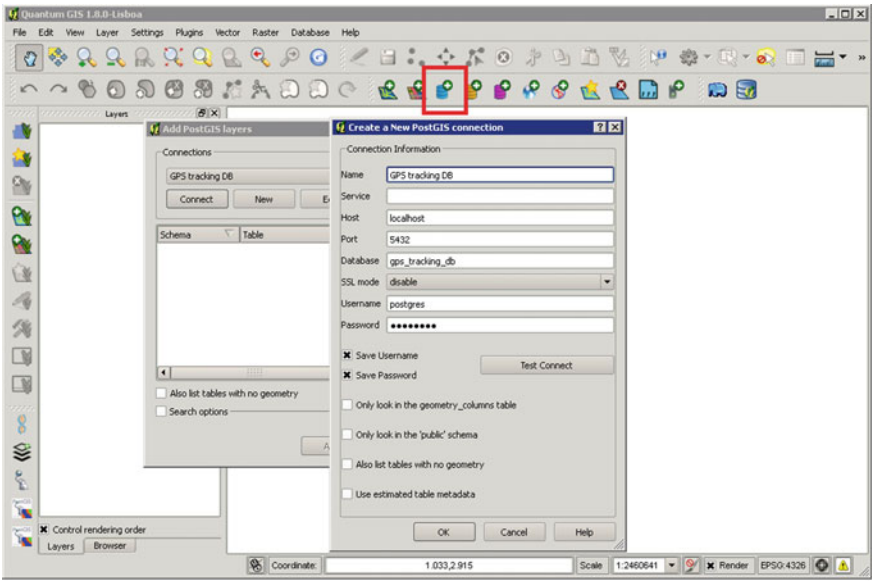

Fig. 5.1 Connection to the database from QGIS

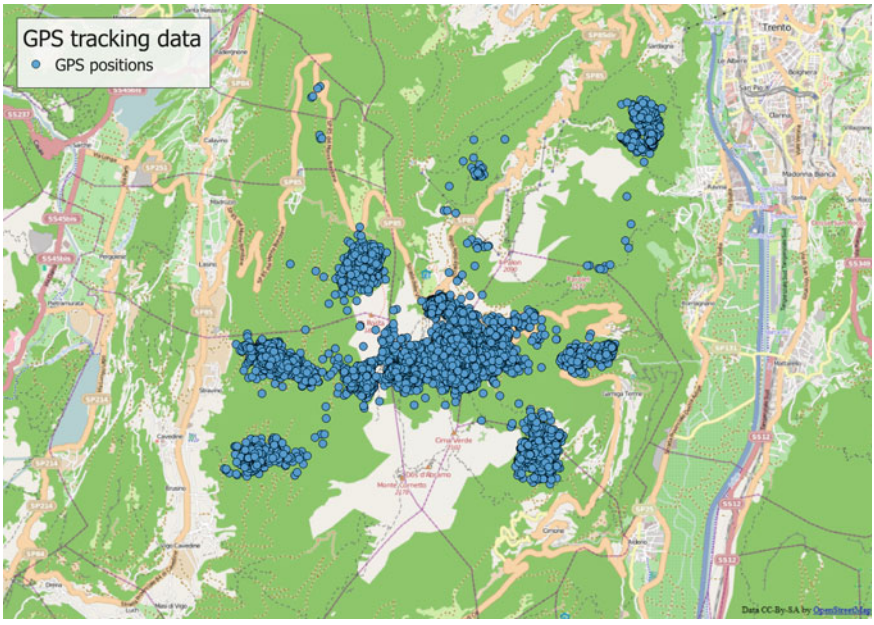

Fig. 5.2 GPS positions visualised in QGIS, zoomed in on the study area to exclude outliers

without making a copy or snapshot of the data and is especially useful when working with dynamic information that is frequently changing.

Automating the Creation of Points from GPS Coordinates

You can automate the population of the *geometry* column so that whenever a new GPS position is uploaded in the table *main.gps_data_animals*, the spatial geometry is also created. To do so, you need a trigger and its related function. Here is the SQL code to generate the function:

```
CREATE OR REPLACE FUNCTION tools.new_gps_data_animals()
RETURNS trigger AS
$BODY$
DECLARE
  thegeom geometry;
BEGIN

IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
  thegeom = ST_SetSRID(ST_MakePoint(NEW.longitude, NEW.latitude),4326);
  NEW.geom = thegeom;
END IF;

RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.new_gps_data_animals()
IS 'When called by a trigger (insert_gps_locations) this function populates
the field geom using the values from longitude and latitude fields.';
```

And here is the SQL code to generate the trigger:

```
CREATE TRIGGER insert_gps_location
  BEFORE INSERT
  ON main.gps_data_animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.new_gps_data_animals();
```

You can see the result by deleting all the records from the *main.gps_data_animals* table, e.g. for animal 2, and reloading them. As you have set an automatic procedure to synchronise *main.gps_data_animals* table with the information contained in the table *main.gps_sensors_animals*, you can drop the record related to animal 2 from *main.gps_sensors_animals,* and this will affect *main.gps_data_animals* in a cascade effect (note that it will not affect the original data in *main.gps_data*):

```
DELETE FROM
  main.gps_sensors_animals
WHERE
  animals_id = 2;
```

There are now no rows for animal 2 in the table *main.gps_data_animals*. You can verify this by retrieving the number of locations per animal:

```
SELECT
  animals_id, count(animals_id)
FROM
  main.gps_data_animals
GROUP BY
  animals_id
ORDER BY
  animals_id;
```

The result should be

```
 animals_id | count
------------+-------
          1 |  2114
          3 |  2106
          4 |  2869
          5 |  2924
```

Note that animal 2 is not in the list. Now, you reload the record in the *main.gps_sensors_animals*:

```
INSERT INTO main.gps_sensors_animals
  (animals_id, gps_sensors_id, start_time, end_time, notes)
VALUES
  (2,1,'2005-03-20 16:03:14 +0','2006-05-27 17:00:00 +0','End of battery life.
  Sensor not recovered.');
```

You can see that records have been readded to *main.gps_data_animals* by reloading the original data stored in *main.gps_data*, with the *geometry* field correctly and automatically populated (when longitude and latitude are not null):

```
SELECT
  animals_id, count(animals_id) AS num_records, count(geom) AS
  num_records_valid
FROM
  main.gps_data_animals
GROUP BY
  animals_id
ORDER BY
  animals_id;
```

The result is

```
animals_id | num_records | num_records_valid
-----------+-------------+------------------
         1 |        2114 |              1650
         2 |        2624 |              2196
         3 |        2106 |              1828
         4 |        2869 |              2642
         5 |        2924 |              2696
```

You can now play around with your spatial data set. For example, when you have a number of locations per animal, you can find the centroid of the area covered by the locations:

```
SELECT
  animals_id,
  ST_AsEWKT(
    ST_Centroid(
     ST_Collect(geom))) AS centroid
FROM
  main.gps_data_animals
WHERE
  geom IS NOT NULL
GROUP BY
  animals_id
ORDER BY
  animals_id;
```

The result is

```
animals_id |                        centroid
-----------+-----------------------------------------------------
         1 | SRID=4326;POINT(11.056405072 46.0065913348485)
         2 | SRID=4326;POINT(11.0388902698087 46.0118316898451)
         3 | SRID=4326;POINT(11.062054399453 46.0229784057986)
         4 | SRID=4326;POINT(11.0215063307722 46.0046905791446)
         5 | SRID=4326;POINT(11.0287071960312 46.0085975505935)
```

In this case, you used the SQL command *ST_Collect*[12]. This function returns a *GEOMETRYCOLLECTION* or a *MULTI* object from a set of geometries. The collect function is an 'aggregate' function in the terminology of PostgreSQL. This means that it operates on rows of data, in the same way the *sum* and *mean* functions do. *ST_Collect* and *ST_Union*[13] are often interchangeable. *ST_Collect* is in general orders of magnitude faster than *ST_Union* because it does not try to dissolve boundaries. It merely rolls up single geometries into *MULTI* and *MULTI*

[12] http://postgis.refractions.net/docs/ST_Collect.html.

[13] http://postgis.refractions.net/docs/ST_Union.html.

or mixed geometry types into *Geometry Collections*. The contrary of *ST_Collect* is *ST_Dump*[14], which is a set-returning function.

Creating Spatial Database Views

Special Topic: **PostgreSQL views**

Views are queries permanently stored in the database. For users (and client applications), they work like normal tables, but their data are calculated at query time and not physically stored. Changing the data in a table alters the data shown in subsequent invocations of related views. Views are useful because they can represent a subset of the data contained in a table; can join and simplify multiple tables into a single virtual table; take very little space to store, as the database contains only the definition of a view (i.e. the SQL query), not a copy of all the data it presents; and provide extra security, limiting the degree of exposure of tables to the outer world. On the other hand, a view might take some time to return its data content. For complex computations that are often used, it is more convenient to store the information in a permanent table.

You can create views where derived information is (virtually) stored. First, create a new schema where all the analysis can be accommodated:

```
CREATE SCHEMA analysis
  AUTHORIZATION postgres;
  GRANT USAGE ON SCHEMA analysis TO basic_user;
COMMENT ON SCHEMA analysis
IS 'Schema that stores key layers for analysis.';
ALTER DEFAULT PRIVILEGES
  IN SCHEMA analysis
  GRANT SELECT ON TABLES
  TO basic_user;
```

You can see below an example of a view in which just (spatially valid) positions of a single animal are included, created by joining the information with the animal and lookup tables.

```
CREATE VIEW analysis.view_gps_locations AS
  SELECT
    gps_data_animals.gps_data_animals_id,
    gps_data_animals.animals_id,
    animals.name,
    gps_data_animals.acquisition_time at time zone 'UTC' AS time_utc,
    animals.sex,
    lu_age_class.age_class_description,
    lu_species.species_description,
    gps_data_animals.geom
```

[14] http://postgis.refractions.net/docs/ST_Dump.html.

```
  FROM
    main.gps_data_animals,
    main.animals,
    lu_tables.lu_age_class,
    lu_tables.lu_species
  WHERE
    gps_data_animals.animals_id = animals.animals_id AND
    animals.age_class_code = lu_age_class.age_class_code AND
    animals.species_code = lu_species.species_code AND
    geom IS NOT NULL;
COMMENT ON VIEW analysis.view_gps_locations
IS 'GPS locations.';
```

Although the best way to visualise this view is in a GIS environment (in QGIS, you might need to explicitly define the unique identifier of the view, i.e. *gps_data_animals_id*), you can query its non-spatial content with

```
SELECT
  gps_data_animals_id AS id,
  name AS animal,
  time_utc,
  sex,
  age_class_description AS age,
  species_description AS species
FROM
  analysis.view_gps_locations
LIMIT 10;
```

The result is something similar to

```
 id |  animal   |      time_utc       | sex |  age   | species
----+-----------+---------------------+-----+--------+----------
 62 | Agostino | 2005-03-20 16:03:14 | m   | adult | roe deer
 64 | Agostino | 2005-03-21 00:03:06 | m   | adult | roe deer
 65 | Agostino | 2005-03-21 04:01:45 | m   | adult | roe deer
 67 | Agostino | 2005-03-21 12:02:19 | m   | adult | roe deer
 68 | Agostino | 2005-03-21 16:01:12 | m   | adult | roe deer
 69 | Agostino | 2005-03-21 20:01:49 | m   | adult | roe deer
 70 | Agostino | 2005-03-22 00:01:24 | m   | adult | roe deer
 71 | Agostino | 2005-03-22 04:02:51 | m   | adult | roe deer
 72 | Agostino | 2005-03-22 08:03:04 | m   | adult | roe deer
 73 | Agostino | 2005-03-22 12:01:42 | m   | adult | roe deer
```

Now, you create view with a different representation of your data sets. In this case, you derive a trajectory from GPS points. You have to order locations per animal and per acquisition time; then, you can group them (animal by animal) in a trajectory (stored as a view):

```
CREATE VIEW analysis.view_trajectories AS
  SELECT
    animals_id,
    ST_MakeLine(geom)::geometry(LineString,4326) AS geom
  FROM
    (SELECT animals_id, geom, acquisition_time
    FROM main.gps_data_animals
    WHERE geom IS NOT NULL
    ORDER BY
    animals_id, acquisition_time) AS sel_subquery
  GROUP BY
    animals_id;
COMMENT ON VIEW analysis.view_trajectories
IS 'GPS locations - Trajectories.';
```

In Fig. 5.3, you can see *analysis.view_trajectories* visualised in QGIS.

Lastly, create another view to spatially summarise the GPS data set using convex hull polygons (or minimum convex polygons):

```
CREATE VIEW analysis.view_convex_hulls AS
  SELECT
    animals_id,
    (ST_ConvexHull(ST_Collect(geom)))::geometry(Polygon,4326) AS geom
  FROM
    main.gps_data_animals
  WHERE
    geom IS NOT NULL
  GROUP BY
    animals_id
  ORDER BY
    animals_id;
COMMENT ON VIEW analysis.view_convex_hulls
IS 'GPS locations - Minimum convex polygons.';
```

The result is represented in Fig. 5.4, where you can clearly see the effect of the outliers located far from the study area. Outliers will be filtered out in Chap. 8.

This last view is correct only if the GPS positions are located in a relatively small area (e.g. less than 50 km) because the minimum convex polygon of points in geographic coordinates cannot be calculated assuming that coordinates are related to Euclidean space. At the moment, the function *ST_ConvexHull* does not support the *geography* data type, so the correct way to proceed would be to project the GPS locations in a proper reference system, calculate the minimum convex polygon and then convert the result back to geographic coordinates. In the example, the error is negligible.

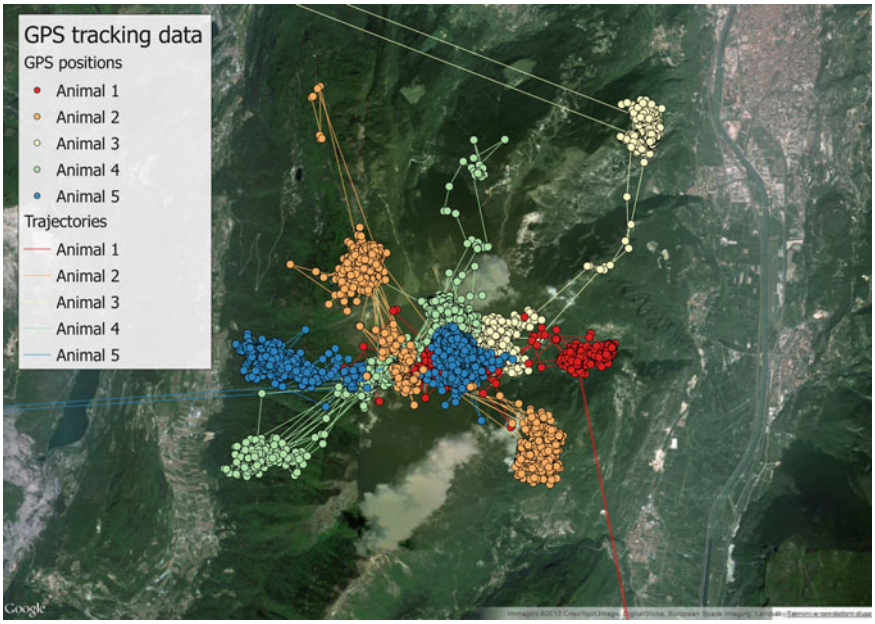

Fig. 5.3 Visualisation of the view with the trajectories (zoom on the study area)

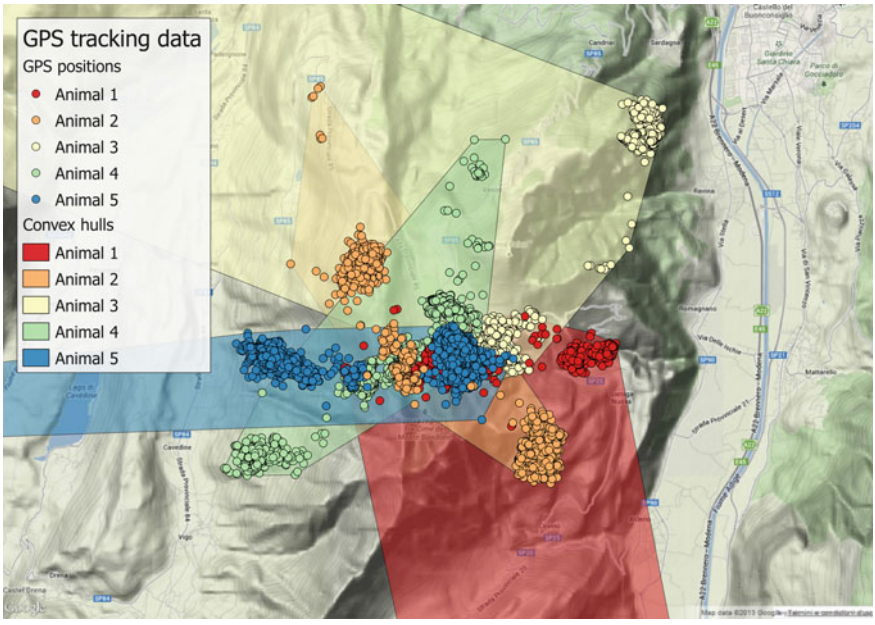

Fig. 5.4 Visualisation of the view with MCP (zoom on the study area)

Vector Data Import and Export

There are different ways to import a shapefile. Compared to the use of tabular data, e.g. in .csv format, the procedure is even easier because users do not have to create an empty table before loading the data. The existing import tools do this job automatically (although in this way you lose control over the data type definition). In QGIS there are two plugins that support shapefile import into PostGIS. The QGIS plugin 'PostGIS Manager' can do the job with a drag-and-drop procedure. Together with the PostGIS installation, a useful tool is automatically created: 'PostGIS Shapefile Import/Export Manager' (located in the PostGIS installation folder). The same kind of tool can also be called from within pgAdmin (in the 'plugin' menu). The same result can be achieved using *shp2pgsql*[15], a command line tool. If the original file uses some specific encoding with characters not supported by standard encoding, the option '*-W*' can be used to solve the problem. Another way to import shapefiles is with the GDAL/OGR[16] library. In general, with any tool, when you load the data, you have to correctly define the reference system, the target table name and the target schema. If the original layer has errors (e.g. overlapping or open polygons, little gaps between adjacent features) it might not be correctly imported and in any case it will probably generate errors when used in PostGIS. Therefore, we strongly recommend that you control the data quality before importing layers into your database.

If you want to export your (vector) spatial layer stored in the database, the easiest way is to load the layer in a GIS environment (e.g. QGIS, ArcGIS) and then simply export to shapefile from there. The 'PostGIS Manager' plugin in QGIS offers advanced tools to perform this task. You can export part of a table or a processed data set using an SQL statement instead of just the name of the table. You can also use the tools mentioned above for data import (*pgsql2shp*, GDAL/OGR library (*ogr2ogr*), PostGIS Shapefile Import/Export Manager).

Connection from Client Applications

One of the main advantages of storing and managing your data in a central database is that you can avoid exporting and importing your data back and forth between different programs, formats or files. Some client applications commonly used in connection with PostgreSQL/PostGIS have specific tools to establish the link with the database (e.g. pgAdmin, QGIS, ArcGIS 10.x). However, as you are likely using different programs to analyse your data, you need to be able to access the database from all of them. This problem can be solved using the Open DataBase Connection (OBDC) which is a protocol to connect to a database in a

[15] http://www.bostongis.com/pgsql2shp_shp2pgsql_quickguide_20.bqg.

[16] http://www.gdal.org/ogr2ogr.html.

standardised way independent from programming languages, database systems and operating systems. ODBC works as a kind of universal translator layer between your program and the database. Virtually any program that is able to handle data today supports ODBC in one way or another.

In Chap. 10, you will see how to use a PosgreSQL/PostGIS database in connection with R.

Special Topic: **Create an ODBC driver in MS Windows**

In the Windows operating system, you can easily create an ODBC connection to your PostgreSQL database. First, you have to install the PostgreSQL ODBC driver on your computer[17]. Then you go to 'Control Panel—Date Sources (ODBC)' or 'Microsoft ODBC Administrator' (according to Windows version), select 'System DSN' tag and click 'Add')[18]. Select 'PostgreSQL Unicode' and the appropriate version (32 or 64 bit, according to PostgreSQL and ODBC Administrator versions), and fill the form with the proper connection parameters. You can check whether it works by clicking 'Test', then click 'Save'. Now you have the ODBC connection available as system DSN. 'Data Source' is the name that identifies the ODBC driver to your database. Once created, you can access your database by calling the ODBC driver through its name. You can test your ODBC connection by connecting the database from, e.g. MS Excel. Sometimes a spreadsheet is useful to produce simple graphics or to use functions that are specific to this kind of tool. To connect to your database data you have to create a connection to a table. Open Excel and select 'Data—Connection' and then 'Add'. Click on the name of the ODBC driver that you created and select the table you want to open in MS Excel. Go to 'Data—Existing connections' and select the connection that you just established. You'll be asked where to place this data. You can choose the existing worksheet or specify a new worksheet. Take your decision and press OK. Now you have your database data visualised in an Excel spreadsheet. Spatial data are visualised as binary data format, and therefore they cannot be properly 'read'. If you want to see the coordinates of the geometry behind, you can use a PostGIS function like *ST_AsText* or *ST_AsEWKT*. Tables are linked to the database. Any change in Excel will not affect the database, but you can refresh the table in Excel by getting the latest version of the linked tables.

References

Corti P, Mather SV, Kraft TJ, Park B (2014) PostGIS Cookbook. Packt Publishing LTD., Birmingham, UK

Obe OR, Hsu LS (2011) PostGIS in action. Manning Publications Company, Greenwich

[17] http://www.postgresql.org/ftp/odbc/versions/msi/.

[18] This process might vary according to the Windows version.

Chapter 6
From Points to Habitat: Relating Environmental Information to GPS Positions

Ferdinando Urbano, Mathieu Basille and Pierre Racine

Abstract Animals move in and interact with complex environments that can be characterised by a set of spatial layers containing environmental data. Spatial databases can manage these different data sets in a unified framework, defining spatial and non-spatial relationships that simplify the analysis of the interaction between animals and their habitat. A large set of analyses can be performed directly in the database with no need for dedicated GIS or statistical software. Such an approach moves the information content managed in the database from a 'geographical space' to an 'animal's ecological space'. This more comprehensive model of the animals' movement ecology reduces the distance between physical reality and the way data are structured in the database, filling the semantic gap between the scientist's view of biological systems and its implementation in the information system. This chapter shows how vector and raster layers can be included in the database and how you can handle them using (spatial) SQL. The database built so far in Chaps. 2, 3, 4 and 5 is extended with environmental ancillary data sets and with an automated procedure to intersect these layers with GPS positions.

Keywords PostGIS · Raster data · Data management · Spatial database

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, 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_6,

Introduction

Animals move in and interact with complex environments that can be characterised by a set of spatial layers containing environmental data. In traditional information systems for wildlife tracking data management, position data are stored in some file-based spatial format (e.g. shapefile). With a multi-steps process in a GIS environment, position data are associated with a set of environmental attributes through an analytical stage (e.g. intersection of GPS positions with vector and raster environmental layers). This process is usually time-consuming and prone to error, implies data replication and often has to be repeated for any new analysis. It also generally involves different tools for vector and raster data. An advanced data management system should achieve the same result with an efficient (and, if needed, automated) procedure, possibly performed as a real-time routine management task. To do so, the first step is to integrate both position data and spatial ancillary information on the environment in a unique framework. This is essential to exploring the animals' behaviour and understanding the ecological relationships that can be revealed by tracking data. Spatial databases can manage these different data sets in a unified framework, defining spatial and non-spatial relationships that simplify the analysis of the interaction between animals and their habitat. A large set of analyses can be performed directly in the database with no need for dedicated GIS or statistical software. This also affects performance, as databases are optimised to run simple processes on large data sets like the ones generated by GPS sensors. Database tools such as triggers and functions can be used, for example, to automatically intersect positions with the ancillary information stored as raster and vector layers. The result is that positions are transformed from a simple pair of numbers (coordinates) to complex multi-dimensional (spatial) objects that define the individual and its habitat in time and space, including their interactions and dependencies. In an additional step, position data can also be joined to activity data to define an even more complete picture of the animal's behaviour (see Chap. 12). Such an approach moves the information content managed in the database from a 'geographical space' to an 'animal's ecological space'. This more comprehensive model of the animal movement ecology reduces the distance between physical reality and the way data are structured in the database, filling the semantic gap between the scientist's view of biological systems and its implementation in the information system. This is not only interesting from a conceptual point of view, but also has deep practical implications. Scientists and wildlife managers can deal with data in the same way they model the object of their study as they can start their analyses from objects that represent the animals in their habitat (which previously was the result of a long and complex process). Moreover, users can directly query these objects using a simple and powerful language (SQL) that is close to their natural language. All these elements strengthen the opportunity provided by GPS data to move from mainly testing statistical hypotheses to focusing on biological hypotheses. Scientists can store, access and manipulate their data in a simple and quick way, which

allows them to formulate biological questions that previously were almost impossible to answer for technical reasons.

This chapter shows how vector and raster layers can be included in the database, how you can handle them using (spatial) SQL and how you can associate with the GPS locations. In the next chapter, we will focus on raster time series using remote sensing images.

Adding Ancillary Environmental Layers

In the exercise, you will see how to integrate a number of spatial features (see Fig. 6.1).

- Points: meteorological stations (derived from MeteoTrentino[1]).
- Linestrings: roads network (derived from OpenStreetMap[2]).
- Polygons: administrative units (derived from ISTAT[3]) and the study area.
- Rasters: land cover (source: Corine[4]) and digital elevation models (source: SRTM[5], see also Jarvis et al. 2008).

Each species and study have specific data sets required and available, so the goal of this example is to show a complete set of procedures that can be replicated and customised on different data sets. When layers are integrated into the database, you can visualise and explore them in a GIS environment (e.g. QGIS).

Once data are loaded into the database, you will extend the *gps_data_animals* table with the environmental attributes derived from the ancillary layers provided in the test data set. You will also modify the function *tools.new_gps_data_animals* to compute these values automatically. In addition, you are encouraged to develop your own (spatial) queries (e.g. detect how many times each animal crosses a road, calculate how many times two animals are in the same place at the same time).

It is a good practice to store your environmental layers in a dedicated schema in order to keep a clear database structure. Let us create the schema *env_data*:

```
CREATE SCHEMA env_data
  AUTHORIZATION postgres;
GRANT USAGE ON SCHEMA env_data TO basic_user;
COMMENT ON SCHEMA env_data
IS 'Schema that stores environmental ancillary information.';
ALTER DEFAULT PRIVILEGES IN SCHEMA env_data
  GRANT SELECT ON TABLES TO basic_user;
```

[1] Provincia autonoma di Trento—Servizio Prevenzione Rischi—Ufficio Previsioni e pianificazione, http://www.meteotrentino.it.

[2] http://www.openstreetmap.org.

[3] http://www.istat.it/it/strumenti/cartografia.

[4] http://www.eea.europa.eu/data-and-maps/data/corine-land-cover-2006-clc2006-100-m-version-12-2009.

[5] http://srtm.csi.cgiar.org/.

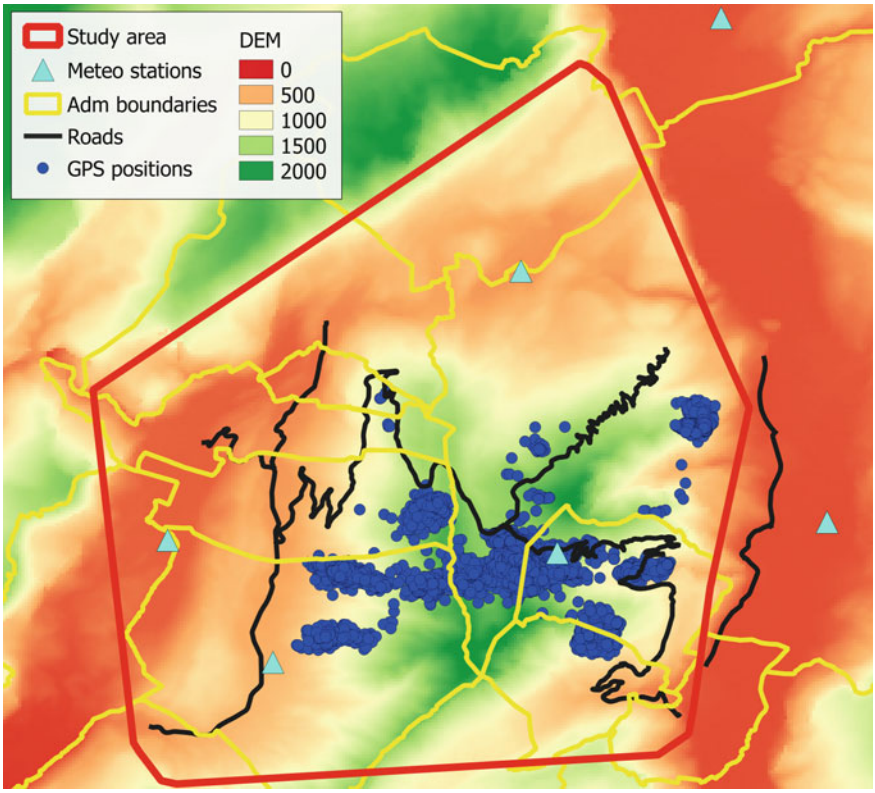

Fig. 6.1 Environmental layers that will be integrated into the database

Importing Shapefiles: Points, Lines and Polygons

Now you can start importing the shapefiles of the (vector) environmental layers included in the test data set. As discussed in Chap. 5, an option is to use the drag-and-drop function of 'DB Manager' (from QGIS Browser) plugin in QGIS (see Fig. 6.2).

Alternatively, a standard solution to import shapefiles (vector data) is the *shp2pgsql* tool. *shp2pgsql* is an external command-line tool, which cannot be run in an SQL interface as it can for a regular SQL command. The code below has to be run in a command-line interpreter (if you are using Windows as operating system, it is also called Command Prompt or MS-DOS shell, see Fig. 6.3). You will see other examples of external tools that are run in the same way, and it is very important to understand the difference between these and SQL commands. In this guide, this difference is represented graphically by white text boxes (see below) for shell commands, while the SQL code is shown in grey text boxes. Start with the meteorological stations:

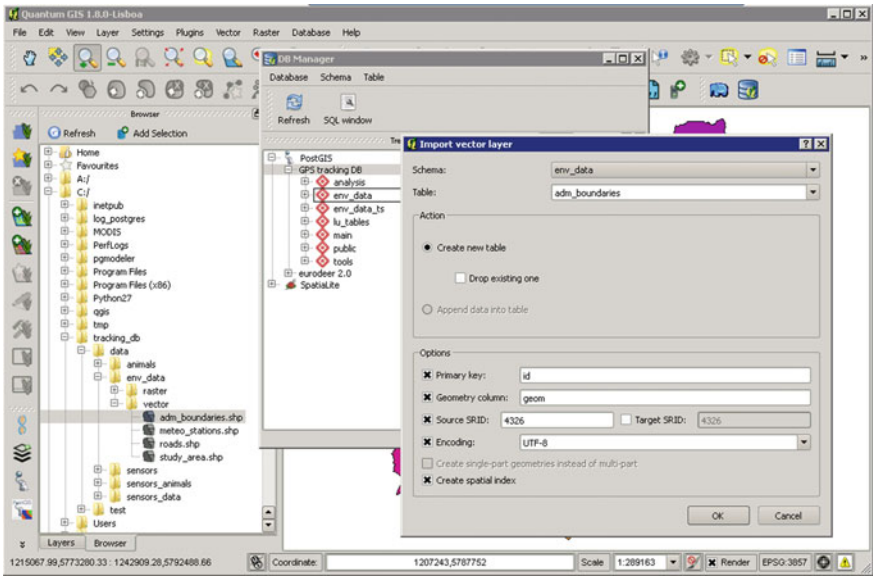

Fig. 6.2 Loading data into PostgreSQL using the drag-and-drop tool in QGIS

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\meteo_stations.shp
env_data.meteo_stations | "C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p
5432 -d gps_tracking_db -U postgres -h localhost
```

Note that the path to *shp2pgsql.exe* and *psql.exe* can be different according to the folder where you installed your version of PostgreSQL. If you connect with the database remotely, you also have to change the address of the server (*-h* option). In the parameters, set the reference system (option *-s*) and create a spatial index for the new table (option *-I*). The result of *shp2pgsql* is a text file with the SQL that generates and populates the table *env_data.meteo_stations*. With the symbol 'I' you 'pipe' (send directly) the SQL to the database (through the PostgreSQL interactive terminal *psql*[6]) where it is automatically executed. You have to set the port (*-p*), the name of the database (*-d*), the user (*-U*), and the password, if requested. In this way, you complete the whole process with a single command. You can refer to *shp2pgsql* documentation for more details. You might have to add the whole path to *psql* and *shp2pgsql*. This depends on the folder where you installed PostgreSQL. You can easily verify the path searching for these two files. You also have to check that the path of your shapefile (meteo_stations.shp) is properly defined.

You can repeat the same operation for the study area layer:

[6] http://www.postgresql.org/docs/9.2/static/app-psql.html.

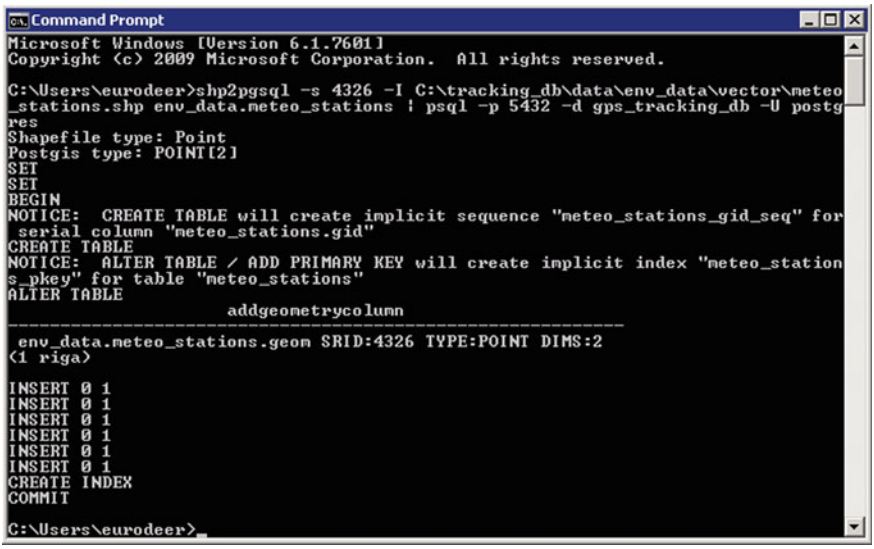

Fig. 6.3 The command shp2pgsql from Windows command-line interpreter

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\study_area.shp env_data.study_area |
"C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p 5432 -d gps_tracking_db -U
postgres -h localhost
```

Next for the roads layer

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\roads.shp env_data.roads | "C:\Program
Files\PostgreSQL\9.2\bin\psql.exe" -p 5432 -d gps_tracking_db -U postgres -h
localhost
```

And for the administrative boundaries

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\adm_boundaries.shp
env_data.adm_boundaries | "C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p
5432 -d gps_tracking_db -U postgres -h localhost
```

Now the shapefiles are in the database as new tables (one table for each shapefile). You can visualise them through a GIS interface (e.g. QGIS). You can also retrieve a summary of the information from all vector layers available in the database with the following command:

```
SELECT * FROM geometry_columns;
```

Importing Raster Files

The primary method to import a raster layer is the command-line tool *raster2pgsql*[7], the equivalent of *shp2pgsql*, but for raster files, that converts GDAL-supported rasters into SQL suitable for loading into PostGIS. It is also capable of loading folders of raster files.

Special Topic: **GDAL**

GDAL[8] (Geospatial Data Abstraction Library) is a (free) library for reading, writing and processing raster geospatial data formats. It has a lot of simple but very powerful and fast command-line tools for raster data translation and processing. The related OGR library provides a similar capability for simple vector data features. GDAL is used by most of the spatial open source tools and by a large number of commercial software programs as well. You will probably benefit in particular from the tools gdalinfo[9] (get a layer's basic metadata), gdal_translate[10] (change data format, change data type, cut), gdalwarp[11] (mosaicing, reprojection and warping utility).

An interesting feature of *raster2pgsql* is its capability to store the rasters inside the database (in-db) or keep them as (out-db) files in the file system (with the *raster2pgsql -R* option). In the last case, only rasters as metadata are stored in the database, not pixel values themselves. Loading out-db rasters as metadata is much faster than loading them completely in the database. Most operations at the pixel values level (e.g. *ST_SummaryStats*) will have equivalent performance with out- and in-db rasters. Other functions, like *ST_Tile*, involving only the metadata, will be faster with out-db rasters. Another advantage of out-db rasters is that they stay accessible for external applications unable to query databases (with SQL). However, the administrator must make sure that the link between what is in the db (the path to the raster file in the file system) is not broken (e.g. by moving or renaming the files). On the other hand, only in-db rasters can be generated with *CREATE TABLE* and modified with *UPDATE* statements. Which is the best choice depends on the size of the data set and on considerations about performance and database management. A good practice is generally to load very large raster data sets as out-db and to load smaller ones as in-db to save time on loading and to avoid repeatedly backing up huge, static rasters.

The QGIS plugin 'Load Raster to PostGIS' can also be used to import raster data with a graphical interface. An important parameter to set when importing raster layers is the number of tiles (*-t* option). Tiles are small subsets of the image and correspond to a physical record in the table. This approach dramatically

[7] http://postgis.net/docs/manual-2.0/using_raster.xml.html#RT_Raster_Loader.

[8] http://www.gdal.org/.

[9] http://www.gdal.org/gdalinfo.html.

[10] http://www.gdal.org/gdal_translate.html.

[11] http://www.gdal.org/gdalwarp.html.

decreases the time required to retrieve information. The recommended values for the tile option range from 20 × 20 to 100 × 100. Here is the code (to be run in the Command Prompt) to transform a raster (the digital elevation model derived from SRTM) into the SQL code that is then used to physically load the raster into the database (as you did with *shp2pgsql* for vectors):

```
"C:\Program Files\PostgreSQL\9.2\bin\raster2pgsql.exe" -I -M -C -s 4326 -t
20x20 C:\tracking_db\data\env_data\raster\srtm_dem.tif env_data.srtm_dem |
"C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p 5432 -d gps_tracking_db -U
postgres -h localhost
```

You can repeat the same process on the land cover layer:

```
"C:\Program Files\PostgreSQL\9.2\bin\raster2pgsql.exe" -I -M -C -s 3035
-t 20x20 C:\tracking_db\data\env_data\raster\corine06.tif
env_data.corine_land_cover | "C:\Program Files\PostgreSQL\9.2\bin\psql.exe"
-p 5432 -d gps_tracking_db -U postgres -h localhost
```

The reference system of the Corine land cover data set is not geographic coordinates (SRID 4326), but ETRS89/ETRS-LAEA (SRID 3035), an equal-area projection over Europe. This must be specified with the -s option and kept in mind when this layer will be connected to other spatial layers stored in a different reference system. As with *shp2pgsql.exe*, the *-I* option will create a spatial index on the loaded tiles, speeding up many spatial operations, and the *-C* option will generate a set of constraints on the table, allowing it to be correctly listed in the *raster_columns* metadata table. The land cover raster identifies classes that are labelled by a code (an integer). To specify the meaning of the codes, you can add a table where they are described. In this example, the land cover layer is taken from the Corine project[12]. Classes are described by a hierarchical legend over three nested levels. The legend is provided in the test data set in the file 'corine_legend.csv'. You import the table of the legend (first creating an empty table, and then loading the data):

```
CREATE TABLE env_data.corine_land_cover_legend(
  grid_code integer NOT NULL,
  clc_l3_code character(3),
  label1 character varying,
  label2 character varying,
  label3 character varying,
  CONSTRAINT corine_land_cover_legend_pkey
    PRIMARY KEY (grid_code ));
COMMENT ON TABLE env_data.corine_land_cover_legend
IS 'Legend of Corine land cover, associating the numeric code to the three
nested levels.';
```

[12] http://www.eea.europa.eu/publications/COR0-landcover.

Then, you load the data:

```
COPY env_data.corine_land_cover_legend
FROM
  'C:\tracking_db\data\env_data\raster\corine_legend.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

You can retrieve a summary of the information from all raster layers available in the database with the following command:

```
SELECT * FROM raster_columns;
```

To keep a well-documented database, add comments to describe all the spatial layers that you have added:

```
COMMENT ON TABLE env_data.adm_boundaries
IS 'Layer (polygons) of administrative boundaries (comuni).';
COMMENT ON TABLE env_data.corine_land_cover
IS 'Layer (raster) of land cover (from Corine project).';
COMMENT ON TABLE env_data.meteo_stations
IS 'Layer (points) of meteo stations.';
COMMENT ON TABLE env_data.roads
IS 'Layer (lines) of roads network.';
COMMENT ON TABLE env_data.srtm_dem
IS 'Layer (raster) of digital elevation model (from SRTM project).';
COMMENT ON TABLE env_data.study_area
IS 'Layer (polygons) of the boundaries of the study area.';
```

Querying Spatial Environmental Data

As the set of ancillary (spatial) information is now loaded into the database, you can start playing with this information using spatial SQL queries. In fact, it is possible with spatial SQL to run queries that explicitly handle the spatial relationships among the different spatial tables that you have stored in the database. In the following examples, SQL statements will show you how to take advantage of PostGIS features to manage, explore, and analyse spatial objects, with optimised performances and no need for specific GIS interfaces. You start by asking for the name of the administrative unit (‘comune’, Italian commune) in which the point at coordinates (11, 46) (longitude, latitude) is located. There are two commands that are used when it comes to intersection of spatial elements: *ST_Intersects* and *ST_Intersection*. The former returns *true* if two features intersect, while the latter returns the geometry produced by the intersection of the objects. In this case, *ST_Intersects* is used to select the right commune:

```
SELECT
  nome_com
FROM
  env_data.adm_boundaries
WHERE
  ST_Intersects((ST_SetSRID(ST_MakePoint(11,46), 4326)), geom);
```

The result is

```
nome_com
----------
Cavedine
```

In the second example, you compute the distance (rounded to the metre) from the point at coordinates (11, 46) to all the meteorological stations (ordered by distance) in the table *env_data.meteo_stations*. This information could be used, for example, to derive the precipitation and temperature for a GPS position at the given acquisition time, weighting the measurement from each station according to the distance from the point. In this case, *ST_Distance_Spheroid* is used. Alternatively, you could use *ST_Distance* and cast your geometries as *geography* data types.

```
SELECT
  station_id, ST_Distance_Spheroid((ST_SetSRID(ST_MakePoint(11,46), 4326)),
  geom, 'SPHEROID["WGS 84",6378137,298.257223563]')::integer AS distance
FROM
  env_data.meteo_stations
ORDER BY
  distance;
```

The result is

```
station_id | distance
-----------+----------
         1 |     2224
         2 |     4080
         5 |     4569
         4 |    10085
         3 |    10374
         6 |    18755
```

In the third example, you compute the distance to the closest road:

```
SELECT
  ST_Distance((ST_SetSRID(ST_MakePoint(11,46), 4326))::geography,
  geom::geography)::integer AS distance
FROM
  env_data.roads
ORDER BY
  distance
LIMIT 1;
```

The result is

```
 distance
----------
     1560
```

For users, the data type (vector, raster) used to store spatial information is not so relevant when they query their data: queries should transparently use any kind of spatial data as input. Users can then focus on the environmental model instead of worrying about the data model. In the next example, you intersect a point with two raster layers (altitude and land cover) in the same way you do for vector layers. In the case of land cover, the point must first be projected into the Corine reference system (SRID 3035). In the raster layer, just the Corine code class (integer) is stored while the legend is stored in the table *env_data.corine_land_cover_legend*. In the query, the code class is joined to the legend table and the code description is returned. This is an example of integration of both spatial and non-spatial elements in the same query.

```
SELECT
  ST_Value(srtm_dem.rast,
  (ST_SetSRID(ST_MakePoint(11,46), 4326))) AS altitude,
  ST_value(corine_land_cover.rast,
  ST_transform((ST_SetSRID(ST_MakePoint(11,46), 4326)), 3035)) AS land_cover,
  label2,label3
FROM
  env_data.corine_land_cover,
  env_data.srtm_dem,
  env_data.corine_land_cover_legend
WHERE
  ST_Intersects(corine_land_cover.rast,
    ST_Transform((ST_SetSRID(ST_MakePoint(11,46), 4326)), 3035)) AND
  ST_Intersects(srtm_dem.rast,(ST_SetSRID(ST_MakePoint(11,46), 4326))) AND
  grid_code = ST_Value(corine_land_cover.rast,
    ST_Transform((ST_SetSRID(ST_MakePoint(11,46), 4326)), 3035));
```

The result is

```
 altitude | land_cover | label2  |      label3
----------+------------+---------+----------------
      956 |         24 | Forests | Coniferous forest
```

Now, combine roads and administrative boundaries to compute how many metres of roads there are in each administrative unit. You first have to intersect the two layers (*ST_Intersection*), then compute the length (*ST_Length*) and summarise per administrative unit (*sum* associated with *GROUP BY* clause).

```
SELECT
  nome_com,
  sum(ST_Length(
    (ST_Intersection(roads.geom, adm_boundaries.geom))::geography))::integer
    AS total_length
FROM
  env_data.roads,
  env_data.adm_boundaries
WHERE
  ST_Intersects(roads.geom, adm_boundaries.geom)
GROUP BY
  nome_com
ORDER BY
  total_length desc;
```

The result of the query is

```
   nome_com    | total_length
---------------+--------------
 Trento        |        24552
 Lasino        |        15298
 Garniga Terme |        12653
 Calavino      |         6185
 Cavedine      |         5802
 Cimone        |         5142
 Padergnone    |         4510
 Vezzano       |         1618
 Aldeno        |         1367
```

The last examples are about the interaction between rasters and polygons. In this case, you compute some statistics (minimum, maximum, mean and standard deviation) for the altitude within the study area:

```
SELECT
  (sum(ST_Area(((gv).geom)::geography)))/1000000 area,
  min((gv).val) alt_min,
  max((gv).val) alt_max,
  avg((gv).val) alt_avg,
  stddev((gv).val) alt_stddev
FROM
  (SELECT
    ST_intersection(rast, geom) AS gv
  FROM
    env_data.srtm_dem,
    env_data.study_area
  WHERE
    ST_intersects(rast, geom)
) foo;
```

The result, from which it is possible to appreciate the large variability of altitude across the study area, is

```
       area        | alt_min | alt_max |      alt_avg       |    alt_stddev
-------------------+---------+---------+--------------------+------------------
 199.018552456188 |     180 |    2133 | 879.286157704969 | 422.56622698974
```

You might also be interested in the number of pixels of each land cover type within the study area. As with the previous example, you first intersect the study area with the raster of interest, but in this case, you need to reproject the study area polygon into the coordinate system of the Corine land cover raster (SRID 3035). With the following query, you can see the dominance of mixed forests in the study area:

```
SELECT (pvc).value, SUM((pvc).count) AS total, label3
FROM
  (SELECT ST_ValueCount(rast) AS pvc
  FROM env_data.corine_land_cover, env_data.study_area
  WHERE ST_Intersects(rast, ST_Transform(geom, 3035))) AS cnts,
  env_data.corine_land_cover_legend
WHERE grid_code = (pvc).value
GROUP BY (pvc).value, label3
ORDER BY (pvc).value;
```

The result is

```
 lc_class | total |                    label3
----------+-------+-----------------------------------------------
        1 |   114 | Continuous urban fabric
        2 |   817 | Discontinuous urban fabric
        3 |   324 | Industrial or commercial units
        7 |   125 | Mineral extraction sites
       16 |   324 | Fruit trees and berry plantations
       18 |   760 | Pastures
       19 |   237 | Annual crops associated with permanent crops
       20 |  1967 | Complex cultivation patterns
       21 |  2700 | Land principally occupied by agriculture
       23 |  4473 | Broad-leaved forest
       24 |  2867 | Coniferous forest
       25 |  8762 | Mixed forest
       26 |   600 | Natural grasslands
       27 |   586 | Moors and heathland
       29 |  1524 | Transitional woodland-shrub
       31 |   188 | Bare rocks
       32 |   611 | Sparsely vegetated areas
       41 |   221 | Water bodies
```

The previous query can be modified to return the percentage of each class over the total number of pixels. This can be achieved using window functions[13]:

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

```
SELECT
  (pvc).value,
  (SUM((pvc).count)*100/ SUM(SUM((pvc).count)) over ())::numeric(4,2)
   AS total_perc, label3
FROM
  (SELECT ST_ValueCount(rast) AS pvc
   FROM env_data.corine_land_cover, env_data.study_area
   WHERE ST_Intersects(rast, ST_Transform(geom, 3035))) AS cnts,
   env_data.corine_land_cover_legend
WHERE grid_code = (pvc).value
GROUP BY (pvc).value, label3
ORDER BY (pvc).value;
```

The result is

```
 value | total_perc |                    label3
-------+------------+-----------------------------------------------
     1 |       0.42 | Continuous urban fabric
     2 |       3.00 | Discontinuous urban fabric
     3 |       1.19 | Industrial or commercial units
     7 |       0.46 | Mineral extraction sites
    16 |       1.19 | Fruit trees and berry plantations
    18 |       2.79 | Pastures
    19 |       0.87 | Annual crops associated with permanent crops
    20 |       7.23 | Complex cultivation patterns
    21 |       9.93 | Land principally occupied by agriculture
    23 |      16.44 | Broad-leaved forest
    24 |      10.54 | Coniferous forest
    25 |      32.21 | Mixed forest
    26 |       2.21 | Natural grasslands
    27 |       2.15 | Moors and heathland
    29 |       5.60 | Transitional woodland-shrub
    31 |       0.69 | Bare rocks
    32 |       2.25 | Sparsely vegetated areas
    41 |       0.81 | Water bodies
```

Associate Environmental Characteristics with GPS Locations

After this general introduction to the use of spatial SQL to explore spatial layers, you can now use these tools to associate environmental characteristics with GPS positions. You can find a more extended introduction to spatial SQL in Obe and Hsu (2011). The goal here is to automatically transform position data from simple points to objects holding information about the habitat and conditions where the animals were located at a certain moment in time. You will use the points to automatically extract, by the mean of an SQL trigger, this information from other ecological layers. The first step is to add the new fields of information into the

main.gps_data_animals table. You will add columns for the name of the administrative unit to which the GPS position belongs, the code for the land cover it is located in, the altitude from the digital elevation model (which can then be used as the third dimension of the point), the id of the closest meteorological station and the distance to the closest road:

```
ALTER TABLE main.gps_data_animals
  ADD COLUMN pro_com integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN corine_land_cover_code integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN altitude_srtm integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN station_id integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN roads_dist integer;
```

These are several common examples of environmental information that can be associated with GPS positions, and others can be implemented according to specific needs. It is important to keep in mind that these spatial relationships are implicitly determined by the coordinates of the elements involved; you do not necessarily have to store these values in a table as you can compute them on the fly whenever you need. Moreover, you might need different information according to the specific study (e.g. the land cover composition in an area of 1 km around each GPS position instead of the value of the pixel where the point is located). Computing these spatial relationships on the fly can require significant time, so in some cases, it is preferable to run the query just once and permanently store the most relevant parameters for your specific study (think about what you will most likely use often). Another advantage of making the relations explicit within tables is that you can then create indexes on columns of these tables. This is not possible with on-the-fly sub-queries. Making many small queries and hence creating many tables and indexing them along the way is generally more efficient in terms of processing time than trying to do everything in a long and complex query. This is not necessarily true when the data set is small enough, as indexes are mostly efficient on large tables. Sometimes, the time necessary to write many SQL statements and the associated indexes exceed the time necessary to execute them. In that case, it might be more efficient to write a single, long and complex statement and forget about the indexes. This does not apply to the following trigger function, as all the ecological layers were well indexed at load time and it does not rely on intermediate sub-queries of those layers.

The next step is to implement the computation of these parameters inside the automated process of associating GPS positions with animals (from *gps_data* to *gps_data_animals*). To achieve this goal, you have to modify the trigger function *tools.new_gps_data_animals*. In fact, the function *tools.new_gps_data_animals* is activated whenever a new location is inserted into *gps_data_animals* (from *gps_data*). It adds new information (i.e. fills additional fields) to the incoming

record (e.g. creates the geometry object from latitude and longitude values) before it is uploaded into the *gps_data_animals* table in the code, *NEW* is used to reference the new record not yet inserted). The SQL code that does this is below. The drawback of this function is that it will slow down the import of a large set of positions at once (e.g. millions or more), but it has little impact when you manage a continuous data flow from sensors, even for a large number of sensors deployed at the same time.

```
CREATE OR REPLACE FUNCTION tools.new_gps_data_animals()
RETURNS trigger AS
$BODY$
DECLARE
  thegeom geometry;
BEGIN

IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
  thegeom = ST_SetSRID(ST_MakePoint(NEW.longitude, NEW.latitude), 4326);
  NEW.geom =thegeom;
  NEW.pro_com =
    (SELECT pro_com::integer
     FROM env_data.adm_boundaries
     WHERE ST_Intersects(geom,thegeom));
  NEW.corine_land_cover_code =
    (SELECT ST_Value(rast,ST_Transform(thegeom,3035))
     FROM env_data.corine_land_cover
     WHERE ST_Intersects(ST_Transform(thegeom,3035), rast));
  NEW.altitude_srtm =
    (SELECT ST_Value(rast,thegeom)
     FROM env_data.srtm_dem
     WHERE ST_Intersects(thegeom, rast));
  NEW.station_id =
    (SELECT station_id::integer
     FROM env_data.meteo_stations
     ORDER BY ST_Distance_Spheroid(thegeom, geom, 'SPHEROID["WGS 84",
     6378137,298.257223563]')
     LIMIT 1);
  NEW.roads_dist =
    (SELECT ST_Distance(thegeom::geography, geom::geography)::integer
     FROM env_data.roads
     ORDER BY ST_distance(thegeom::geography, geom::geography)
     LIMIT 1);
END IF;

RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.new_gps_data_animals()
IS 'When called by the trigger insert_gps_positions (raised whenever a new
position is uploaded into gps_data_animals) this function gets the longitude
and latitude values and sets the geometry field accordingly, computing a set
of derived environmental information calculated intersecting or relating the
position with the environmental ancillary layers.';
```

As the trigger function is run during GPS data import, the function only works on the records that are imported after it was created, and not on data imported previously. To see the effects, you have to add new positions or delete and reload the GPS positions stored in *gps_data_animals.* You can do this by saving the records in *gps_sensors_animals* in an external .csv file, and then deleting the records from the table (which also deletes the records in *gps_data_animals* in a cascade effect). When you reload them, the new function will be activated by the trigger that was just defined, and the new attributes will be calculated. You can perform these steps with the following commands.

First, check how many records you have per animal:

```
SELECT animals_id, count(animals_id)
FROM main.gps_data_animals
GROUP BY animals_id;
```

The result is

```
 animals_id | count
------------+-------
          4 |  2869
          5 |  2924
          2 |  2624
          1 |  2114
          3 |  2106
```

Then, copy the table *main.gps_sensors_animals* into an external file.

```
COPY
  (SELECT animals_id, gps_sensors_id, start_time, end_time, notes
   FROM main.gps_sensors_animals)
TO
  'c:/tracking_db/test/gps_sensors_animals.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

You then delete all the records in *main.gps_sensors_animals*, which will delete (in a cascade) all the records in *main.gps_data_animals.*

```
DELETE FROM main.gps_sensors_animals;
```

You can verify that there are now no records in *main.gps_data_animals* (the query should return 0 rows).

```
SELECT * FROM main.gps_data_animals;
```

The final step is to reload the .csv file into *main.gps_sensors_animals*. This will launch the trigger functions that recreate all the records in *main.gps_data_animals*,

in which the fields related to environmental attributes are also automatically updated. Note that, due to the different triggers that imply massive computations, the query can take several minutes to execute[14].

```
COPY main.gps_sensors_animals
  (animals_id, gps_sensors_id, start_time, end_time, notes)
FROM
  'c:/tracking_db/test/gps_sensors_animals.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

You can verify that all the fields are updated:

```
SELECT
  gps_data_animals_id AS id, acquisition_time, pro_com,
  corine_land_cover_code AS lc_code, altitude_srtm AS alt, station_id AS
  meteo, roads_dist AS dist
FROM
  main.gps_data_animals
WHERE
  geom IS NOT NULL
LIMIT 10;
```

The result is

```
  id   |    acquisition_time    | pro_com | lc_code | alt  | meteo | dist
-------+------------------------+---------+---------+------+-------+------
 15275 | 2005-10-23 22:00:53+02 |   22091 |      18 | 1536 |     5 |  812
 15276 | 2005-10-24 02:00:55+02 |   22091 |      18 | 1519 |     5 |  740
 15277 | 2005-10-24 06:00:55+02 |   22091 |      18 | 1531 |     5 |  598
 15280 | 2005-10-24 18:02:57+02 |   22091 |      23 | 1198 |     5 |  586
 15281 | 2005-10-24 22:01:49+02 |   22091 |      25 | 1480 |     5 |  319
 15282 | 2005-10-25 02:01:23+02 |   22091 |      18 | 1531 |     5 |  678
 15283 | 2005-10-25 06:00:53+02 |   22091 |      18 | 1521 |     5 |  678
 15284 | 2005-10-25 10:01:10+02 |   22091 |      23 | 1469 |     5 |  546
 15285 | 2005-10-25 14:01:26+02 |   22091 |      23 | 1412 |     5 |  571
 15286 | 2005-10-25 18:02:29+02 |   22091 |      23 | 1435 |     5 |  465
```

You can also check that all the records of every animal are in *main.gps_data_animals*:

```
SELECT animals_id, count(*) FROM main.gps_data_animals GROUP BY animals_id;
```

[14] You can skip this step and speedup the process by simply calculating the environmental attributes with an update query.

The result is

```
 animals_id | count
------------+-------
          4 |  2869
          5 |  2924
          2 |  2624
          1 |  2114
          3 |  2106
```

As you can see, the whole process can take a few minutes, as you are calculating the environmental attributes for the whole data set at once. As discussed in the previous chapters, the use of triggers and indexes to automatise data flow and speedup analyses might imply processing times that are not sustainable when large data sets are imported at once. In this case, it might be preferable to update environmental attributes and calculate indexes in a later stage to speed up the import process. In this book, we assume that in the operational environment where the database is developed, the data flow is continuous, with large but still limited data sets imported at intervals. You can compare this processing time with what is generally required to achieve the same result in a classic GIS environment based on flat files (e.g. shapefiles, .tif). Do not forget to consider that you can use these minutes for a coffee break, while the database does the job for you, instead of clicking here and there in your favourite GIS application!

References

Jarvis A, Reuter HI, Nelson A, Guevara E (2008) Hole-filled seamless SRTM data V4. International centre for tropical agriculture (CIAT)

Obe OR, Hsu LS (2011) PostGIS in action. Manning Publications Company, Greenwich

Chapter 7
Tracking Animals in a Dynamic Environment: Remote Sensing Image Time Series

Mathieu Basille, Ferdinando Urbano, Pierre Racine, Valerio Capecchi and Francesca Cagnacci

Abstract This chapter looks into the spatiotemporal dimension of both animal tracking data sets and the dynamic environmental data that can be associated with them. Typically, these geographic layers derive from remote sensing measurements, commonly those collected by sensors deployed on earth-orbiting satellites, which can be updated on a monthly, weekly or even daily basis. The modelling potential for integrating these two levels of ecological complexity (animal movement and environmental variability) is huge and comes from the possibility to investigate processes as they build up, i.e. in a full dynamic framework. This chapter's exercise will describe how to integrate dynamic environmental data in the spatial database and join to animal locations one of the most used indices for ecological productivity and phenology, the normalised difference vegetation index (NDVI) derived from MODIS. The exercise is based on the database built so far in Chaps. 2, 3, 4, 5 and 6.

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

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

V. Capecchi
Istituto di Biometeorologia, Consiglio Nazionale delle Ricerche,
Via Madonna del piano 10 50019 Sesto Fiorentino, Firenze, Italy
e-mail: capecchi@lamma.rete.toscana.it

F. Cagnacci
Biodiversity and Molecular Ecology Department, Research and Innovation Centre,
Fondazione Edmund Mach, via E. Mach 1, 38010 S.Michele all'Adige, TN, Italy
e-mail: francesca.cagnacci@fmach.it

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_7,

Keywords NDVI · Raster time series · Spatial database · Spatiotemporal intersection

Introduction

The advancement in movement ecology from a data perspective can reach its full potential only by combining the technology of animal tracking with the technology of other environmental sensing programmes (Cagnacci et al. 2010). Ecology is fundamentally spatial, and animal ecology is obviously no exception (Turchin 1998). Any scientific question in animal ecology cannot overlook its spatial dimension, and in particular the dynamic interaction between individual animals or populations, and the environment in which the ecological processes occur. Movement provides the mechanistic link to explain this complex ecosystem interaction, as the movement path is dynamically determined by external factors, through their effect on the individual's state and the life history characteristics of an animal (Nathan et al. 2008). Therefore, most modelling approaches for animal movement include environmental factors as explanatory variables. As illustrated in earlier portions of this book, this technically implies the intersection of animal locations with environmental layers, in order to extract the information about the environment that is embedded in spatial coordinates. It appears very clear at this stage, though, that animal locations are not only spatial, but are also fully defined by spatial and temporal coordinates (as given by the acquisition time).

Logically, the same temporal definition also applies to environmental layers. Some characteristics of the landscape, such as land cover or road networks, can be considered static over a large period of time (of the order of several years), and these static environmental layers are commonly intersected with animal locations to infer habitat use and selection by animals (e.g. Resource Selection Functions, RSF, Manly et al. 2002). However, many characteristics relevant to wildlife, such as vegetation biomass or road traffic, are indeed subject to temporal variability (of the order of hours to weeks) in the landscape and would be better represented by dynamic layers that correspond closely to the conditions actually encountered by an animal moving across the landscape (Moorcroft 2012). In this case, using static environmental layers directly limits the potential of wildlife tracking data, reduces the power of inference of statistical models and sometimes even introduces sources of bias (Basille et al. 2013).

Nowadays, satellite-based remote sensing can provide dynamic global coverage of medium-resolution images that can be used to compute a large number of environmental parameters very useful to wildlife studies. Through remote sensing, it is possible to acquire spatial time series which can then be linked to animal locations, fully exploiting the spatiotemporal nature of wildlife tracking data. Numerous satellites and other sensor networks can now provide information on resources on a monthly, weekly or even daily basis, which can be used as explanatory variables in statistical models (e.g. Pettorelli et al. 2006) or to

parameterise Bayesian inferences or mechanistic models. One of the most commonly used satellite-derived environmental time series is the normalised difference vegetation index (NDVI) but other examples include data sets on ocean primary productivity, surface temperature or salinity, all available in equally fine spatial and temporal scales (McClain 2009), and, in North America, snow depth data at daily scales (SNODAS, at 1-km resolution), spatial temperature and precipitation at monthly scales (PRISM data model, at 1-km resolution), and meteorological data on wind and pressure (ESRL[1]). Snow cover, NDVI and sea surface temperature are some examples of indices that can be used as explanatory variables in statistical models (e.g. Pettorelli et al. 2006) or to parameterise Bayesian inferences or mechanistic models. Moreover, there are user-friendly spatial tools to acquire (LPDAAC NASA website 2008[2]) and process (e.g. Marine Geospatial Ecology Tools—MGET[3], a plugin for the proprietary software ESRI ArcGIS and the free Movebank tool Env-DATA System[4]) data from the moderate-resolution imaging spectroradiometer (MODIS), a major provider of NDVI.

The main shortcoming of such remote sensing layers is the relatively low spatial resolution (e.g. 250 m for MODIS, e.g. Cracknell 1997; 1 km for SPOT vegetation, e.g. Maisongrande et al. 2004), which does not fit the current average bias of wildlife tracking GPS locations (less than 20 m, see Frair et al. 2010 for a review), thus potentially leading to a spatial mismatch between the animal-based information and the environmental layers (note that the resolution can still be perfectly fine, depending on the overall spatial variability and the species and biological process under study). Yet, this is much more desirable than using static layers when the temporal variability is an essential component of the ecological inference (Basille et al. 2013). Higher resolution images and new types of information (e.g. forest structure) are presently provided by new types of sensors, such as those from LIDAR, radar or hyper-spectral remote sensing technology. However, the use of these images for intersection with wildlife tracking data sets is still limited by the high cost of source data, including direct costs such as flights of aircrafts, that restricts the collection of comprehensive high-resolution time series. In the case of animals that move over large distances (regions, nations, continents), these limitations are even greater. Few software packages provide the functionalities to handle time series of images easily (Eerens et al. 2014), while at the same time offering a complete set of the tools required by movement ecology in general and tracking data in particular.

In this chapter, we discuss the integration in the spatial database of one of the most used indices for ecological productivity and phenology, i.e. NDVI, derived from MODIS images. The intersection of NDVI images with GPS locations requires a

[1] http://www.esrl.noaa.gov/.

[2] https://lpdaac.usgs.gov/.

[3] http://mgel.env.duke.edu/mget.

[4] http://www.movebank.org/node/6607.

system that is able to handle large amounts of data and explicitly manage both spatial and temporal dimensions, which makes PostGIS an ideal candidate for the task.

MODIS NDVI Data Series

The MODIS instrument operates on NASA's Terra and Aqua spacecraft. The instrument views the entire earth's surface every 1–2 days and captures data in 36 spectral bands ranging in wavelengths from 0.4 to 14.4 μm and at varying spatial resolutions (250 m, 500 m and 1 km). The global MODIS vegetation indices (MODIS 13 products, MODIS 1999) are designed to provide consistent spatial and temporal comparisons of vegetation conditions. Red and near-infrared reflectances, centred at 645 and 858 nm, respectively, are used to determine vegetation indices, including the well-known NDVI, at daily, 8 d, 16 d and monthly scales. This index is calculated by contrasting intense chlorophyll pigment absorption in the red against the high reflectance of leaf mesophyll in the near infrared. It is a proxy of plant photosynthetic activity and has been found to be highly related to the green leaf area index (LAI) and to the fraction of photosynthetically active radiation absorbed by vegetation (FAPAR; see for instance Bannari et al. 1995).

Past studies have demonstrated the potential of using NDVI data to study vegetation dynamics (Townshend and Justice 1986; Verhoef et al. 1996). More recently, several applications have been developed using MODIS NDVI data such as land cover change detection (Lunetta et al. 2006), monitoring forest phenophases (Yu and Zhuang 2006), modelling wheat yield (Moriondo et al. 2007) and other applications in forest and agricultural sciences. However, the utility of the MODIS NDVI data products is limited by the availability of high-quality data (e.g. cloud free), and several processing steps are required before using the data: acquisition via Web facilities, reprojection from the native sinusoidal projection to a standard latitude–longitude format, eventually the mosaicking of two or more tiles into a single tile. A number of processing techniques to 'smooth' the data and obtain a cleaned (no clouds) time series of NDVI imagery have also been implemented. These kinds of processes are usually based on a set of ancillary information on the data quality of each pixel that is provided together with MODIS NDVI.

In the framework of the present project, a simple R[5] procedure has been adapted in order to download, reproject and mosaic the NDVI data. The procedure is flexible and can be modified to work with other MODIS data (snow, land surface temperature, etc.). It is dependent on the MODIS Reprojection Tool (MRT), a set of tools developed by the NASA to manipulate MODIS files. MRT enables users to read data files in the native HDF-EOS format, specify a geographic subset or specific science data sets as input to processing, perform geographic

[5] http://www.r-project.org/.

transformation to a different coordinate system/cartographic projection and write the output to a GDAL-compatible file format.

A preliminary visual examination of the available NDVI images indicates that they may contain a variable number of pixels with erroneous values. Several techniques have been developed to remove these pseudo-hikes and drops in the time series (probably due to clouds) and substitute the missing data with a reliable value. In the present case, we applied a very simple but efficient procedure, first developed and described by Escadafal et al. (2001), for the 10-day NOAA-AVHRR NDVI images. The procedure was also applied in other similar cases in the European context (Maselli et al. 2006) with the 10-day NOAA-AVHRR and SPOT-VGT NDVI images. Here, we adapted the procedure to work with the 16-day MODIS images. The procedure simply consists of a preliminary filtering in order to remove isolated pixels with anomalous NDVI values and replace them with local (5-point) averages. The final result of such a procedure is shown in Fig. 7.1, which displays NDVI values extracted before and after the smoothing for a location in the municipality of Terlago (46.10°N, 11.05°E, northern Italy). Note that, for convenience purposes, NDVI values have been multiplied by 10,000 to be stored as integers with a maximum value of 10,000 (you will have to keep this in mind when presenting NDVI values).

Dealing with Raster Time Series

Raster time series are quite common from medium- and low-resolution data sets generated by satellites that record information at the same location on earth at regular time intervals. In this case, each pixel has both a spatial and a temporal reference. In this exercise, you integrate an NDVI data set of 92 MODIS images covering the period 2005–2008 (spatial resolution of 1 km and temporal resolution of 16 days). In this example, you will use the *env_data* schema to store raster time series, in order to keep it transparent to the user: all environmental data (static or dynamic) are in this schema. However, over larger amounts of data, it might be useful to store raster time series in a different schema to support an easier and more efficient backup procedure.

When you import a raster image using *raster2pgsql*, a new record is added in the target table for each raster, including one for each tile if the raster was tiled (see Chap. 6). At this point, each record does not consider time yet and is thus simply a spatial object. To transform each record into a spatiotemporal object, you must add a field with the timestamp of the data acquisition, or, better, the time range covered by the data if it is related to a period. The time associated with each raster (and each tile) can usually be derived from the name of the file, where this information is typically embedded. In the case of MODIS composite over 16 days, this is the first day of the 16-day period associated with the image in the form '*MODIS_NDVI_yyyy_mm_dd.tif*' where *yyyy* is the year, *mm* the month, and *dd* the day.

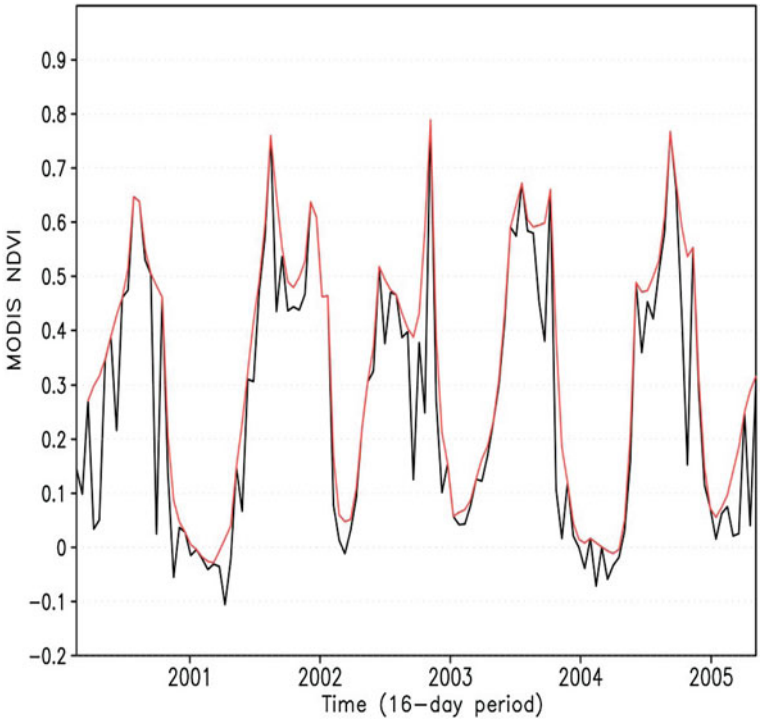

Fig. 7.1 Temporal profiles of raw (*black line*) and cleaned (*red line*) MODIS NDVI data over a 5-year period (2000–2004) for a location in the municipality of Terlago (46.10°N, 11.05°E, northern Italy)

Time Ranges in PostgreSQL

A new data type was introduced in PostgreSQL 9.2 to facilitate storage and manipulation of ranges, i.e. an interval of values with a beginning and an end. The range type[6], stored in a single column, eliminates the need to store the beginning and end values in two separate columns and allows for much simpler comparisons as there is no longer a need to check for both bounds. Ranges can be discrete (e.g. every integer from 0 to 10 included) or continuous (e.g. all real numbers between 0 and 10 included, or any point in time between 1 January and 30 January 2013).

The simplest example would be a series of integers, say from 1 to 10:

```
SELECT int4range(1, 10);
```

[6] http://www.postgresql.org/docs/9.2/static/rangetypes.html.

The result is

```
 int4range
-----------
 [1,10)
```

A range is defined solely by its lower and upper bounds, and, by default, assumes that the lower bound is included, but the upper bound is excluded. In other words, the former query defined the series 1, 2, 3, 4, 5, 6, 7, 8, 9 and did not include 10. As can be seen from the output, the convention used by PostgreSQL is a square bracket for an inclusion and a parenthesis for an exclusion. Each range type has a constructor function with the same name as the range type (e.g. *int4range*, *numrange,* or *daterange*), and accepts as a third argument the specification of the bound (e.g. *'[)'*, which is the default setting). Hence, a range of dates (from 20 March 2013 to 22 September 2013 inclusive) would be constructed like this:

```
SELECT daterange('2013-03-20', '2013-09-22', '[]');
```

The result is

```
        daterange
-------------------------
 [2013-03-20,2013-09-23)
```

Note that in this case, PostgreSQL transformed the result with the default *'[)'* notation, which excludes the upper bound, using the next day. The same result can be achieved by the use of an explicit formulation 'casted' to the range type of interest:

```
SELECT '[2013-03-20, 2013-09-22]'::daterange;
```

This query results in the exact same output, i.e. the period containing all days from 20 March 2013 (included) to 22 September 2013 (included), which defines the period between the two equinoxes when days are longer than nights in the northern hemisphere in 2013. To avoid any confusion, it might be a good practice to use explicit formulas, which forces the declaration of the bounds, and are hence less error prone. You could be even more precise and specify the exact times between the two equinoxes using the *tsrange* type:

```
SELECT '[2013-03-20 11:01:55, 2013-09-22 20:44:08)'::tsrange;
```

The result is

```
                     tsrange
-----------------------------------------------
 ["2013-03-20 11:01:55","2013-09-22 20:44:08")
```

The main interest of using a range is to easily check whether a value (or another range) is included in it. Note that only elements from the same type can be compared (e.g. dates with dates, timestamps with timestamps). You thus need to use explicit casting when necessary. Let us check for instance whether you are currently between the two equinoxes of 2013, using the operator '@>' (containment):

```
SELECT '[2013-03-20, 2013-09-22]'::daterange @> now()::date AS in_range;
```

If you run this query after 22 September 2013, the output of the last query is likely to be false. Other useful operators include equality (=), union (+), intersection (*) or overlap (&&)[7]:

```
SELECT
  '[2013-03-20, 2013-09-22]'::daterange = daterange('2013-03-20',
  '2013-09-22', '[]') AS equal_range;
SELECT
  '[2013-03-20, 2013-09-22]'::daterange + '[2013-06-01, 2014-01-01)'
  ::daterange AS union_range;
SELECT
  '[2013-03-20, 2013-09-22]'::daterange * '[2013-06-01, 2014-01-01)'
  ::daterange AS intersection_range;
SELECT
  '[2013-03-20, 2013-09-22]'::daterange && '[2013-06-01, 2014-01-01)'
  ::daterange AS overlap_range;
```

While the first two return '*t*', the union range is '*[2013-03-20,2014-01-01)*' and the intersection is '*[2013-06-01,2013-09-23)*'.

Finally, note that a range can be infinite on one side and thus does not have a lower or a upper bound. This is achieved by using the NULL bound in the constructor or an empty value in the explicit formulation, for example

```
SELECT
  '[2013-01-01,)'::daterange;
```

or

```
SELECT
  '[2013-01-01,)'::daterange @> now()::date AS after_2013;
```

[7] See the full list of operators and functions here: http://www.postgresql.org/docs/9.2/static/functions-range.html.

Import the Raster Time Series

With this data type, you can now associate each image or tile with the correct time reference, that is, the 16-day period associated with each raster. This will make the spatiotemporal intersection with GPS positions possible by allowing direct comparisons with GPS timestamps.

To start, create an empty table to store the NDVI images, including a field for the temporal reference (of type *daterange*) and its index:

```
CREATE TABLE env_data.ndvi_modis(
  rid serial NOT NULL,
  rast raster,
  filename text,
  acquisition_range daterange,
  CONSTRAINT ndvi_modis_pkey
    PRIMARY KEY (rid));

CREATE INDEX ndvi_modis_wkb_rast_idx
  ON env_data.ndvi_modis
  USING GIST (ST_ConvexHull(rast));

COMMENT ON TABLE env_data.ndvi_modis
IS 'Table that stores values of smoothed MODIS NDVI (16-day periods).';
```

Now, the trick is to use two arguments of the *raster2pgsql* command (see also Chap. 6): *-F* to include the raster file name as a column of the table (which will then be used by the trigger function) and *-a* to append the data to an existing table, instead of creating a new one. Another aspect is the absence of a flagged 'no data' value in the MODIS NDVI files. In these rasters, -3000 represents an empty pixel, but this information is not stored in the raster: you will have to declare it explicitly using the *'-N'* argument. The NDVI data used here consist of 92 tif images from January 2005 to December 2008, but you can import all of them in a single operation using the wildcard character '*' in the input filename. You can thus run the following command in the Command Prompt[8] (warning: you might need to adjust the rasters' path according to your own set-up):

```
C:\Program Files\PostgreSQL\9.2\bin\raster2pgsql.exe -a -C -F -M -s 4326 -t
20x20 -N -3000 C:\tracking_db\data\env_data\raster\raster_ts\*.tif
env_data.ndvi_modis | psql -p 5432 -d gps_tracking_db -U postgres
```

You can confirm that the raster was properly loaded with all its attributes by looking at the *raster_columns* view, which stores raster metadata (here, you only retrieve the table's schema, name, SRID and NoData value, but it is a good practice to examine all information stored in this view):

[8] Note that this is not an a SQL code and cannot be run in an SQL interface.

```
SELECT
  r_table_schema AS schema,
  r_table_name AS table,
  srid,
  nodata_values AS nodata
FROM raster_columns
WHERE r_table_name = 'ndvi_modis';
```

```
 schema   |   table    | srid | nodata
----------+------------+------+--------
env_data | ndvi_modis | 4326 | {-3000}
```

Each raster file embeds the acquisition period in its filename. For instance, 'MODIS_NDVI_2005_01_01.tif' is associated with the period from 1 January 2005 (included) to 17 January 2005 (excluded). As you can see, the period is encoded on 10 characters following the common prefix 'MODIS_NDVI_'. This allows you to use the *substring* function to extract the year, the month and the starting day from the filename (which was automatically stored in the *filename* field during the import). For instance, you can extract the starting date from the first raster imported (which should be *'MODIS_NDVI_2005_01_01.tif'*):

```
SELECT filename,
  (substring(filename FROM 12 FOR 4) || '-' ||
   substring(filename FROM 17 FOR 2) || '-' ||
   substring(filename FROM 20 FOR 2))::date AS start
FROM env_data.ndvi_modis LIMIT 1;
```

```
        filename          |   start
--------------------------+------------
MODIS_NDVI_2005_01_01.tif | 2005-01-01
```

The same approach can be used to define the ending date of the period, by simply adding 16 days to the previous date (remember that this day will be excluded from the range). Note that adding 16 days takes into account the additional day at the end of February in leap years:

```
SELECT filename,
  (substring(filename FROM 12 FOR 4) || '-' ||
   substring(filename FROM 17 FOR 2) || '-' ||
   substring(filename FROM 20 FOR 2))::date + 16 AS end
FROM env_data.ndvi_modis LIMIT 1;
```

```
        filename          |    end
--------------------------+------------
MODIS_NDVI_2005_01_01.tif | 2005-01-17
```

In the case of more complex filenames with a variable number of characters, you could still retrieve the encoded date using the *substring* function, by extracting

the relevant characters relative to some other characters found first using the *position* function. Let us now update the table by converting the filenames into the date ranges according to the convention used in file naming (note that there is an additional constraint that selects 1 January when the start date + 16 days exceeds the beginning of the year):

```
UPDATE env_data.ndvi_modis
SET acquisition_range = daterange(
  (substring(filename FROM 12 FOR 4) || '-' ||
   substring(filename FROM 17 FOR 2) || '-' ||
   substring(filename FROM 20 FOR 2))::date,
  LEAST((substring(filename FROM 12 FOR 4) || '-' ||
         substring(filename FROM 17 FOR 2) || '-' ||
         substring(filename FROM 20 FOR 2))::date + 16,
        (substring(filename FROM 12 FOR 4)::integer + 1
        || '-' || '01' || '-' || '01')::date));
```

As for any type of column, if the table contains a large number of rows (e.g. >10,000), querying based on the *acquisition_range* will be faster if you first index it (you can do it even if the table is not that big, as the PostgreSQL planner will determine whether the query will be faster by using the index or not):

```
CREATE INDEX ndvi_modis_acquisition_range_idx
ON env_data.ndvi_modis (acquisition_range);
```

Now, each tile (and therefore each pixel) has a spatial and a temporal component and thus can be queried according to both criteria. For instance, these are the 10 first tiles corresponding to 1 March 2008, using the '@>' operator ('contains'). Note that this is a leap year so that the corresponding period ends on 5 March:

```
SELECT rid, filename, acquisition_range
FROM env_data.ndvi_modis
WHERE acquisition_range @> '2008-03-01'::date
LIMIT 10;
```

```
 rid  |          filename          |    acquisition_range
------+----------------------------+-------------------------
 3889 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3890 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3891 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3892 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3893 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3894 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3895 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3896 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3897 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
 3898 | MODIS_NDVI_2008_02_18.tif | [2008-02-18,2008-03-05)
```

Based on this, you can now create a trigger and its associated function to automatically create the appropriate date range during the NDVI data import. Note that the *ndvi_acquisition_range_update* function will be activated before an NDVI tile is loaded, so that the transaction is aborted if, for any reason, the *acquisition_range* cannot be computed, and only valid rows are inserted into the *ndvi_modis* table:

```
CREATE OR REPLACE FUNCTION tools.ndvi_acquisition_range_update()
RETURNS trigger AS
$BODY$
BEGIN
  NEW.acquisition_range = daterange(
    (substring(NEW.filename FROM 12 FOR 4) || '-' ||
     substring(NEW.filename FROM 17 FOR 2) || '-' ||
     substring(NEW.filename FROM 20 FOR 2))::date,
    LEAST((substring(NEW.filename FROM 12 FOR 4) || '-' ||
           substring(NEW.filename FROM 17 FOR 2) || '-' ||
           substring(NEW.filename FROM 20 FOR 2))::date + 16,
          (substring(NEW.filename FROM 12 FOR 4)::integer + 1
          || '-' || '01' || '-' || '01')::date));
RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.ndvi_acquisition_range_update()
IS 'This function is raised whenever a new record is inserted into the MODIS
NDVI time series table in order to define the date range. The
acquisition_range value is derived from the original filename (that has the
structure MODIS_NDVI_YYYY_MM_DD.tif)';

CREATE TRIGGER update_ndvi_acquisition_range
BEFORE INSERT ON env_data.ndvi_modis
  FOR EACH ROW EXECUTE PROCEDURE tools.ndvi_acquisition_range_update();
```

Every time you add new NDVI rasters, the *acquisition_range* will then be updated appropriately. At this stage, your database contains all environmental data proposed for the database in this book and should look like Fig. 7.2 (using the DB Manager in QGIS).

Intersection of Locations and NDVI Rasters

To intersect a GPS position with this kind of data set, both temporal and spatial criteria must be defined. In the next example, you retrieve the MODIS NDVI value at point (11, 46) using the *ST_Value* PostGIS SQL function and for the whole year 2005 with the '&&' operator ('overlap'):

```
SELECT
  rid,
  acquisition_range,
  ST_Value(rast, ST_SetSRID(ST_MakePoint(11, 46), 4326)) / 10000 AS ndvi
FROM env_data.ndvi_modis
WHERE ST_Intersects(ST_SetSRID(ST_MakePoint(11, 46), 4326), rast)
  AND acquisition_range && '[2005-01-01,2005-12-31]'::daterange
ORDER BY acquisition_range;
```

The result gives you the complete NDVI profile at this location for the year 2005:

```
 rid  |    acquisition_range    |  ndvi
------+-------------------------+--------
   31 | [2005-01-01,2005-01-17) | 0.7047
   85 | [2005-01-17,2005-02-02) | 0.6397
  139 | [2005-02-02,2005-02-18) | 0.5974
  193 | [2005-02-18,2005-03-06) | 0.5645
  247 | [2005-03-06,2005-03-22) | 0.5745
  301 | [2005-03-22,2005-04-07) | 0.6076
  355 | [2005-04-07,2005-04-23) |  0.649
  409 | [2005-04-23,2005-05-09) | 0.8086
  463 | [2005-05-09,2005-05-25) | 0.8511
  517 | [2005-05-25,2005-06-10) | 0.8935
  571 | [2005-06-10,2005-06-26) | 0.8935
  625 | [2005-06-26,2005-07-12) | 0.8951
  679 | [2005-07-12,2005-07-28) | 0.8979
  733 | [2005-07-28,2005-08-13) | 0.9006
  787 | [2005-08-13,2005-08-29) |  0.907
  841 | [2005-08-29,2005-09-14) | 0.8682
  895 | [2005-09-14,2005-09-30) | 0.8441
  949 | [2005-09-30,2005-10-16) | 0.7556
 1003 | [2005-10-16,2005-11-01) | 0.6895
 1057 | [2005-11-01,2005-11-17) | 0.6979
 1111 | [2005-11-17,2005-12-03) | 0.7291
 1165 | [2005-12-03,2005-12-19) | 0.7778
 1219 | [2005-12-19,2006-01-01) | 0.9654
```

In Fig. 7.3, the NDVI variation for the year is displayed in graphical format (screenshot taken from QGIS).

You can now retrieve NDVI values at coordinates from real animals:

```
SELECT
  animals_id AS ani_id,
  ST_X(geom) AS x,
  ST_Y(geom) AS y,
  acquisition_time,
  ST_Value(rast, geom) / 10000 AS ndvi
FROM main.gps_data_animals, env_data.ndvi_modis
WHERE ST_Intersects(geom, rast)
  AND acquisition_range @> acquisition_time::date
ORDER BY ani_id, acquisition_time
LIMIT 10;
```

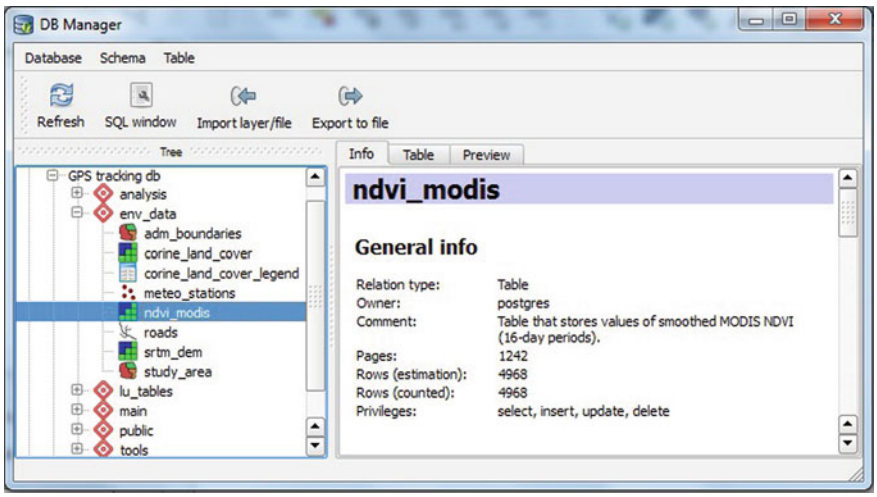

Fig. 7.2 Summary of the different environmental layers using the DB Manager in QGIS, showing the type of data in the database (*lines*, *polygons*, *raster* or *simple tables*)

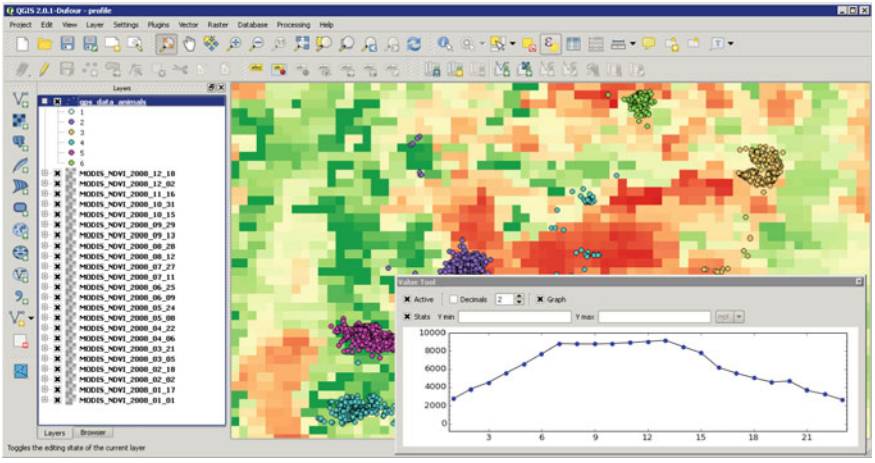

Fig. 7.3 Example of complete NDVI profile for the year 2005 in a pixel in the study area. Remember that NDVI values have been multiplied by 10,000

```
 ani_id |   x    |   y    |     acquisition_time     |  ndvi
--------+--------+--------+--------------------------+--------
      1 | 11.044 | 46.011 | 2005-10-18 16:00:54-04 | 0.5426
      1 | 11.045 | 46.012 | 2005-10-18 20:01:23-04 | 0.5426
      1 | 11.045 | 46.008 | 2005-10-19 00:02:22-04 | 0.6566
      1 | 11.046 | 46.006 | 2005-10-19 04:03:08-04 | 0.5839
      1 | 11.043 | 46.010 | 2005-10-20 16:00:53-04 | 0.5528
      1 | 11.042 | 46.011 | 2005-10-20 20:00:48-04 | 0.5528
      1 | 11.041 | 46.010 | 2005-10-21 00:00:53-04 | 0.5429
      1 | 11.044 | 46.007 | 2005-10-21 04:01:42-04 | 0.6566
      1 | 11.046 | 46.007 | 2005-10-21 12:01:16-04 | 0.6566
      1 | 11.038 | 46.009 | 2005-10-21 16:01:23-04 | 0.5252
```

Now, as an example of the capabilities of PostGIS, let us try to retrieve NDVI values in animal home ranges during a whole season. In habitat selection studies, the habitat used by an individual is generally compared to the habitat that is considered available to the individual. In this example, we assume that the convex polygon encompassing all locations of the winter 2005–2006 defines the area available during this season. In contrast, the exact locations determine what was used by the animals. You will then, for each animal monitored during the winter 2005–2006, compute the average NDVI value at all locations and the average NDVI value in the area covered during the same season. The following query uses the *WITH*[9] syntax, which allows you to break down a seemingly complex query: in this case, you first compute the convex polygons during winter (in *mcp*), then extract the NDVI values in these polygons (in *ndvi_winter_mcp*), and then extract NDVI values at the GPS locations (in *ndvi_winter_locs*). Finally, in the last part of the query, you just display the relevant information:

```
WITH
  mcp AS (
    SELECT
      animals_id,
      min(acquisition_time) AS start_time,
      max(acquisition_time) AS end_time,
      ST_ConvexHull(ST_Collect(geom)) AS geom
    FROM main.gps_data_animals
    WHERE acquisition_time >= '2005-12-21'::date
      AND acquisition_time < '2006-03-21'::date
    GROUP BY animals_id),
  ndvi_winter_mcp AS (
    SELECT
      animals_id,
      start_time,
      end_time,
      ST_SummaryStats(ST_Clip(ST_Union(rast), geom)) AS ss,
```

[9] http://www.postgresql.org/docs/9.2/static/queries-with.html.

```
      acquisition_range
    FROM mcp, env_data.ndvi_modis
    WHERE ST_Intersects(geom, rast)
      AND lower(acquisition_range) >= '2005-12-21'::date
      AND lower(acquisition_range) < '2006-03-21'::date
    GROUP BY animals_id, start_time, end_time, geom,
      acquisition_range),
  ndvi_winter_locs AS (
    SELECT
      animals_id,
      avg(ST_Value(rast, geom)) / 10000 AS mean
    FROM main.gps_data_animals, env_data.ndvi_modis
    WHERE acquisition_time >= '2005-12-21'::date
      AND acquisition_time < '2006-03-21'::date
      AND ST_Intersects(geom, rast)
      AND acquisition_range @> acquisition_time::date
    GROUP BY animals_id)
SELECT
  m.animals_id AS id,
  m.start_time::date,
  m.end_time::date,
  avg((m.ss).mean) / 10000 AS mean_mcp, l.mean AS mean_loc
FROM ndvi_winter_mcp AS m, ndvi_winter_locs AS l
WHERE m.animals_id = l.animals_id
GROUP BY m.animals_id, m.start_time, m.end_time, l.mean
ORDER BY m.animals_id;
```

Note that this complex query takes less than one second! The results indicate that three out of four roe deer actually use greater NDVI values in winter than generally available to them:

```
 id | start_time |  end_time  | mean_mcp | mean_loc
----+------------+------------+----------+----------
  1 | 2005-12-21 | 2006-03-20 |    0.424 |    0.454
  2 | 2005-12-21 | 2006-03-20 |    0.276 |    0.333
  3 | 2005-12-21 | 2006-03-20 |    0.436 |    0.452
  4 | 2005-12-21 | 2006-03-20 |    0.538 |    0.510
```

Automating the Intersection

The last step is now to automate the intersection of the GPS locations and the NDVI data set. The approach is similar to the automatic intersection with other environmental layers (e.g. elevation or land cover) described in Chap. 6; however, the dynamic nature of the NDVI time series makes it slightly more complex. In the case of near real-time monitoring, you will generally acquire GPS data before the NDVI rasters are available. As a consequence, two automated procedures are necessary to update the table *main.gps_data_animals*: one after the import of new GPS locations and one after the import of new NDVI data.

First of all, you need a new column in the *gps_data_animals* table to store the NDVI values:

```
ALTER TABLE main.gps_data_animals
ADD COLUMN ndvi_modis integer;
```

Now, let us first update this column manually for those locations that actually correspond to an NDVI tile in the database:

```
UPDATE
  main.gps_data_animals
SET
  ndvi_modis = ST_Value(rast, geom)
FROM
  env_data.ndvi_modis
WHERE ST_Intersects(geom, rast)
  AND acquisition_range @>
  acquisition_time::date
  AND gps_validity_code = 1 AND
  ndvi_modis IS NULL;
```

You can verify that the fields are updated:

```
SELECT
  gps_data_animals_id AS id, acquisition_time,
  ndvi_modis / 10000.0 AS ndvi
FROM
  main.gps_data_animals
WHERE
  geom IS NOT NULL
ORDER BY
  acquisition_time
LIMIT 10;
```

The result is the following:

```
 id   |    acquisition_time    | ndvi
------+------------------------+-------
39212 | 2005-03-20 11:03:14-05 | 0.447
39214 | 2005-03-20 19:03:06-05 | 0.505
39215 | 2005-03-20 23:01:45-05 | 0.505
39217 | 2005-03-21 07:02:19-05 | 0.431
39218 | 2005-03-21 11:01:12-05 | 0.431
39219 | 2005-03-21 15:01:49-05 | 0.480
39220 | 2005-03-21 19:01:24-05 | 0.480
39221 | 2005-03-21 23:02:51-05 | 0.480
39222 | 2005-03-22 03:03:04-05 | 0.541
39223 | 2005-03-22 07:01:42-05 | 0.541
```

Now, the last, more complicated step, is to use triggers to automate the process. Exactly as in Chap. 6, you need to extend the trigger function *new_gps_data_animals* that will be automatically triggered every time you add new GPS locations to the database:

```
CREATE OR REPLACE FUNCTION tools.new_gps_data_animals()
RETURNS trigger AS
$BODY$
DECLARE
  thegeom geometry;
  thedate date;
BEGIN
IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
  thegeom = ST_SetSRID(ST_MakePoint(NEW.longitude, NEW.latitude), 4326);
  thedate = NEW.acquisition_time::date;
  NEW.geom = thegeom;
  NEW.pro_com =
    (SELECT pro_com::integer
     FROM env_data.adm_boundaries
     WHERE ST_Intersects(geom, thegeom));
  NEW.corine_land_cover_code =
    (SELECT ST_Value(rast,ST_Transform(thegeom, 3035))
     FROM env_data.corine_land_cover
     WHERE ST_Intersects(ST_Transform(thegeom, 3035), rast));
  NEW.altitude_srtm =
    (SELECT ST_Value(rast, thegeom)
     FROM env_data.srtm_dem
     WHERE ST_Intersects(thegeom, rast));
  NEW.station_id =
    (SELECT station_id::integer
     FROM env_data.meteo_stations
     ORDER BY ST_Distance_Spheroid(thegeom, geom, 'SPHEROID["WGS 84",
     6378137,298.257223563]')
     LIMIT 1);
  NEW.roads_dist =
    (SELECT ST_Distance(thegeom::geography, geom::geography)::integer
     FROM env_data.roads
     ORDER BY ST_distance(thegeom::geography, geom::geography)
     LIMIT 1);
  NEW.ndvi_modis =
    (SELECT ST_Value(rast, thegeom)
     FROM env_data.ndvi_modis
     WHERE ST_Intersects(thegeom, rast)
     AND acquisition_range @> thedate);
  END IF;
RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.new_gps_data_animals()
IS 'When called by the trigger insert_gps_positions (raised whenever a new
position is uploaded into gps_data_animals) this function gets the longitude
and latitude values and sets the geometry field accordingly, computing a set
of derived environmental information calculated intersecting or relating the
position with the environmental ancillary layers.';
```

However, note that the update process will be limited by the availability of NDVI data at the time of the GPS data import (NDVI data are generally available two weeks after the period considered, which might then be later than the GPS data import). In order to have a complete database, you thus also need to update the table when new NDVI data are added. You can do it by running the *UPDATE* query every time new images are imported or by creating another trigger function that will automatically update the GPS locations that correspond to the NDVI temporal range in *main.gps_data_animals*. Here is the trigger function:

```
CREATE OR REPLACE FUNCTION tools.ndvi_intersection_update()
RETURNS trigger AS
$BODY$
BEGIN
  UPDATE main.gps_data_animals
  SET ndvi_modis =
    (SELECT ST_Value(NEW.rast, geom)
      FROM env_data.ndvi_modis
      WHERE ST_Intersects(geom, NEW.rast)
      AND NEW.acquisition_range @> NEW.acquisition_time::date)
  WHERE ST_Intersects(geom, NEW.rast)
  AND NEW.acquisition_range @> acquisition_time::date
  AND ndvi.modis IS NULL;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.ndvi_intersection_update ()
IS 'When new NDVI data are added, the ndvi_modis field of
main.gps_data_animals is updated.';
```

The functions will be activated each time new data are loaded into *env_data.ndvi_modis*. The function *ndvi_intersection_update* will be activated after an NDVI tile is loaded, because we want the final version of the NDVI tiles before propagating the updates to other tables. Here is the trigger to achieve this:

```
CREATE TRIGGER update_ndvi_intersection
AFTER INSERT ON env_data.ndvi_modis
  FOR EACH ROW EXECUTE PROCEDURE tools.ndvi_intersection_update();
```

From now on, every time you collect NDVI data and feed them into the database using *raster2pgsql*, the magic will happen!

References

Bannari A, Morin D, Bonn F, Huete AR (1995) A review of vegetation indices. Remote Sens Rev 13:95–120

Basille M, Fortin D, Dussault C, Ouellet JP, Courtois R (2013) Ecologically based definition of seasons clarifies predator-prey interactions. Ecography 36:220–229

Cagnacci F, Boitani L, Powell RA, Boyce MS (2010) Animal ecology meets GPS-based radiotelemetry: a perfect storm of opportunities and challenges. Philos Trans R Soc B: Biol Sci 365:2157–2162

Cracknell AP (1997) The advanced very high resolution radiometer (AVHRR). Taylor & Francis, London

Eerens H, Haesen D, Rembold F, Urbano F, Tote C, Bydekerke L (2014) Image time series processing for agriculture monitoring. Environ Modell Softw 53:154–162

Escadafal R, Bohbot H, Mégier J (2001) Changes in arid mediterranean ecosystems on the long term through earth observation (CAMELEO). Final Report of EU contract IC18-CT97-0155, Edited by Space Applications Institute, JRC, Ispra, Italy

Frair JL, Fieberg J, Hebblewhite M, Cagnacci F, DeCesare NJ, Pedrotti L (2010) Resolving issues of imprecise and habitat biased locations in ecological analyses using GPS telemetry data. Philos Trans R Soc B: Biol Sci 365:2187–2200

Land Processes DAAC (2008) MODIS reprojection tool user's manual. USGS Earth Resources Observation and Science (EROS) Center

Lunetta RS, Knight JF, Ediriwickrema J, Lyon JG, Dorsey Worthy L (2006) Land-cover change detection using multi-temporal MODIS NDVI data. Remote Sens Environ 105:142–154

Maisongrande P, Duchemin B, Dedieu G (2004) VEGETATION/SPOT: an operational mission for the Earth monitoring; presentation of new standard products. Int J Remote Sens 25:9–14

Manly BFJ, McDonald LL, Thomas DL, McDonald TL, Erickson WP (2002) Resource selection by animals. Kluver Academic Publishers, Dordrecht

Maselli F, Barbati A, Chiesi M, Chirici G, Corona P (2006) Use of remotely sensed and ancillary data for estimating forest gross primary productivity in Italy. Remote Sens Environ 100:563–575

McClain CR (2009) A decade of satellite ocean color observations. Annu Rev Marine Sci 1:19–42

MODIS (1999) MODIS Vegetation Index (MOD 13): Algorithm Theoretical Basis Document Page 26 of 29 (version 3)

Moorcroft P (2012) Mechanistic approaches to understanding and predicting mammalian space use: recent advances, future directions. J Mammal 93:903–916

Moriondo M, Maselli F, Bindi M (2007) A simple model of regional wheat yield based on NDVI data. Eur J Agron 26:266–274

Nathan R, Getz WM, Revilla E, Holyoak M, Kadmon R, Saltz D, Smouse PE (2008) A movement ecology paradigm for unifying organismal movement research. Proc Natl Acad Sci 105:19052–19059

Pettorelli N, Gaillard JM, Mysterud A, Duncan P, Stenseth NC, Delorme D, Van Laere G, Toigo C, Klein F (2006) Using a proxy of plant productivity (NDVI) to find key periods for animal performance: the case of roe deer. Oikos 112:565–572

Townshend JRG, Justice CO (1986) Analysis of the dynamics of African vegetation using the normalized difference vegetation index. Int J Remote Sens 8:1189–1207

Turchin P (1998) Quantitative analysis of movement: measuring and modeling population redistribution in plants and animals. Sinauer Associates, Sunderland

Verhoef W, Menenti M, Azzali S (1996) A colour composite of NOAA-AVHRR–NDVI based on time series analysis (1981–1992). Int J Remote Sens 17:231–235

Yu XF, Zhuang DF (2006) Monitoring forest phenophases of Northeast China based on MODIS NDVI data. Resour Sci 28:111–117

Chapter 8
Data Quality: Detection and Management of Outliers

Ferdinando Urbano, Mathieu Basille and Francesca Cagnacci

Abstract Tracking data can potentially be affected by a large set of errors in different steps of data acquisition and processing. Erroneous data can heavily affect analysis, leading to biased inference and misleading wildlife management/conservation suggestions. Data quality assessment is therefore a key step in data management. In this chapter, we especially deal with biased locations, or 'outliers'. While in some cases incorrect data are evident, in many situations, it is not possible to clearly identify locations as outliers because although they are suspicious (e.g. long distances covered by animals in a short time or repeated extreme values), they might still be correct, leaving a margin of uncertainty. In this chapter, different potential errors are identified and a general approach to managing outliers is proposed that tags records rather than deleting them. According to this approach, practical methods to find and mark errors are illustrated on the database created in Chaps. 2, 3, 4, 5, 6 and 7.

Keywords Outlier detection · GPS accuracy · Animal movement · Erroneous data

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

F. Cagnacci
Biodiversity and Molecular Ecology Department, Research and Innovation Centre,
Fondazione Edmund Mach, via E. Mach 1, 38010 S.Michele all'Adige, TN, Italy
e-mail: francesca.cagnacci@fmach.it

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_8,

Introduction

Tracking data can potentially be affected by a large set of errors in different steps of data acquisition and processing, involving malfunctioning or poor performance of the sensor device that may affect measurement, acquisition and recording; failure of transmission hardware or lack of transmission due to network or physical conditions; and errors in data handling and processing. Erroneous location data can substantially affect analysis related to ecological or conservation questions, thus leading to biased inference and misleading wildlife management/conservation conclusions. The nature of positioning error is variable (see Special Topic), but whatever the source and type of errors, they have to be taken into account. Indeed, data quality assessment is a key step in data management.

In this chapter, we especially deal with biased locations or 'outliers'. While in some cases incorrect data are evident, in many situations it is not possible to clearly identify locations as outliers because although they are suspicious (e.g. long distances covered by animals in a short time or repeated extreme values), they might still be correct, leaving a margin of uncertainty. For example, it is evident from Fig. 8.1 that there are at least three points of the GPS data set with clearly incorrect coordinates.

In the exercise presented in this chapter, different potential errors are identified. A general approach to managing outliers is proposed that tags records rather than deleting them. According to this approach, practical methods to find and mark errors are illustrated.

Review of Errors that Can Affect GPS Tracking Data

The following are some of the main errors that can potentially affect data acquired from GPS sensors (points 1 to 5), and that can be classified as GPS location bias, i.e. due to a malfunctioning of the GPS sensor that generates locations with low accuracy (points 6 to 9):

1. Missing records. This means that no information (not even the acquisition time) has been received from the sensor, although it was planned by the acquisition schedule.
2. Records with missing coordinates. In this case, there is a GPS failure probably due to bad GPS coverage or canopy closure. In this case, the information on acquisition time is still valid, even if no coordinates are provided. This corresponds to 'fix rate' error (see Special Topic).
3. Multiple records with the same acquisition time. This has no physical meaning and is a clear error. The main problem here is to decide which record (if any) is correct.
4. Records that contain different values when acquired using different data transfer procedures (e.g. direct download from the sensor through a cable vs. data transmission through the GSM network).

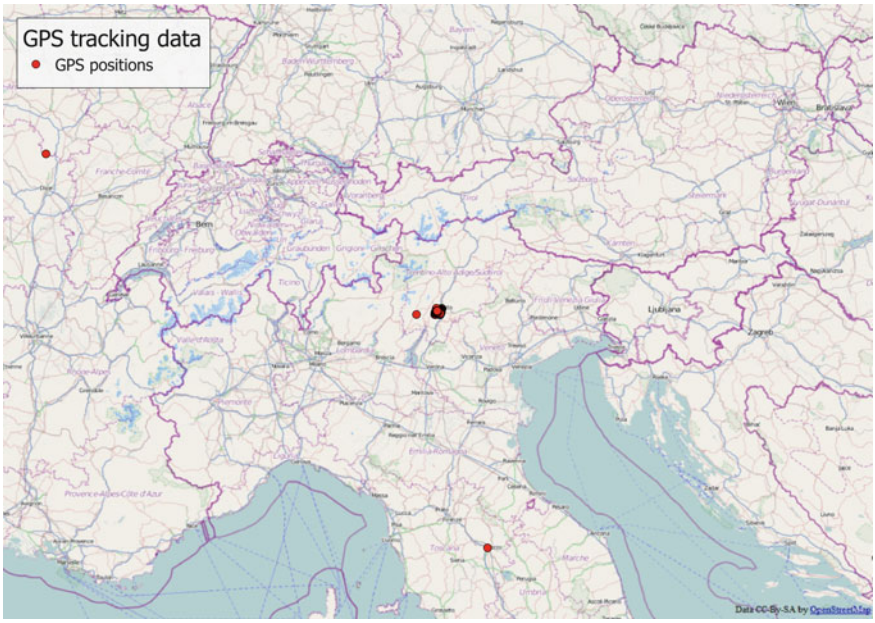

Fig. 8.1 Visualisation of the GPS data set, where three points are evident erroneous positions

5. Records erroneously attributed to an animal because of inexact deployment information. This case is frequent and is usually due to an imprecise definition of the deployment time range of the sensor on the animal. A typical result is locations in the scientist's office followed by a trajectory along the road to the point of capture.
6. Records located outside the study area. In this case, coordinates are incorrect (probably due to malfunctioning of the GPS sensor) and outliers appear very far from the other (valid) locations. This is a special case of impossible movements where the erroneous location is detected even with a simple visual exploration. This can be considered an extreme case of location bias, in terms of accuracy (see Special Topic).
7. Records located in impossible places. This might include (depending on species) sea, lakes or otherwise inaccessible places. Again, the error can be attributed to GPS sensor bias.
8. Records that imply impossible movements (e.g. very long displacements, requiring movement at a speed impossible for the species). In this case, some assumptions on the movement model must be made (e.g. maximum speed).
9. Records that imply improbable movements. In this case, although the movement is physically possible according to the threshold defined, the likelihood of the movement is so low that it raises serious doubts about its reliability. Once the location is tagged as suspicious, analysts can decide whether it should be considered in specific analyses.

GPS sensors usually record other ancillary information that can vary according to vendors and models. Detection of errors in the acquisition of these attributes is not treated here. Examples are the number of satellites used to estimate the position, the dilution of precision (DOP), the temperatures as measured by the sensor associated with the GPS and the altitude estimated by the GPS. Temperature is measured close to the body of the animal, while altitude is not measured on the geoid but as the distance from the centre of the earth: thus in both cases the measure is affected by large errors.

Special Topic: **Dealing with localisation errors associated with the GPS sensor**

A source of uncertainty associated with GPS data is the positioning error of the sensor. GPS error can be classified as bias (i.e. average distance between the 'true location' and the estimated location, where the average value represents the accuracy while the measure of dispersion of repeated measures represents the precision) and fix rate, or the proportion of expected fixes (i.e. those expected according to the schedule of positioning that is programmed on the GPS unit) compared to the number of fixes actually obtained. Both these types of errors are related to several factors, including tag brand, orientation, fix interval (e.g. cold/warm or hot start), and topography and canopy closure. Unfortunately, the relationship between animals and the latter two factors is the subject of a fundamental branch of spatial ecology: habitat selection studies. In extreme synthesis, habitat selection models establish a relationship between the habitat used by animals (estimated by acquired locations) versus available proportion of habitat (e.g. random locations throughout study area or home range). Therefore, a habitat-biased proportion of fixes due to instrumental error may hamper the inferential powers of habitat selection models. A series of solutions have been proposed. For a comprehensive review see Frair et al. (2010). Among others, a robust methodology is the use of spatial predictive models for the probability of GPS acquisition, usually based on dedicated local tests, the so-called Pfix. Data can then be weighted by the inverse of Pfix, so that positions taken in difficult-to-estimate locations are weighted more. In general, it is extremely important to account for GPS bias, especially in resource selection models.

A General Approach to the Management of Erroneous Locations

Once erroneous records are detected, the suggested approach is to keep a copy of all the information as it comes from the sensors (in *gps_data* table), and then mark records affected by each of the possible errors using different tags in the table where locations are associated with animals (*gps_data_animals*). Removing data seems often not so much a problem with GPS data sets, since you probably have thousands of locations anyway. Although keeping incorrect data as valid could be much more of a problem and bias further analyses, suspicious locations, if correct, might be exactly the information needed for a specific analysis (e.g. rutting excursions). The use of a tag to identify the reliability of each record can solve these problems. Records should never be deleted from the data set even when they are completely wrong, for the following reasons:

- If you detect errors with automatic procedures, it is always a good idea to be able to manually check the results to be sure that the method performed as expected, which is not possible if you delete the records.
- If you delete a record, whenever you have to resynchronise your data set with the original source, you will reintroduce the outlier, particularly for erroneous locations that cannot be automatically detected.
- A record can have some values that are wrong (e.g. coordinates), but others that are valid and useful (e.g. timestamp).
- The fact that the record is an outlier is valuable information that you do not want to lose (e.g. you might want to know the success rate of the sensor according to the different types of errors).
- It is important to differentiate missing locations (no data from sensor) from data that were received but erroneous for another reason (incorrect coordinates). As underlined in the Special Topic, the difference between these two types of error is substantial.
- It is often difficult to determine unequivocally that a record is wrong, because this decision is related to assumptions about the species' biology. If all original data are kept, criteria to identify outliers can be changed at any time.
- What looks useless in most cases (e.g. records without coordinates) might be very useful in other studies that were not planned when data were acquired and screened.
- Repeated erroneous data are a fairly reliable clue that a sensor is not working properly, and you might use this information to decide whether and when to replace it.

In the following examples, you will explore the location data set hunting for possible errors. First, you will create a field in the GPS data table where you can store a tag associated with each erroneous or suspicious record. Then, you will define a list of codes, one for each possible type of error. In general, a preliminary visual exploration of the spatial distribution of the entire set of locations can be useful for detecting the general spatial patterns of the animals' movements and evident outlier locations.

To tag locations as errors or unreliable data, you first create a new field (*sensor_validity_code*) in the *gps_data_animals* table. At the same time, a list of codes corresponding to all possible errors must be created using a lookup table *gps_validity*, linked to the *sensor_validity_code* field with a foreign key. When an outlier detection process identifies an error, the record is marked with the corresponding tag code. In the analytical stage, users can decide to exclude all or part of the records tagged as erroneous. The evident errors (e.g. points outside the study area) can be automatically marked in the import procedure, while some other detection algorithms are better run by users when required because they imply a long processing time or might need a fine tuning of the parameters. First, add the new field to the table:

```
ALTER TABLE main.gps_data_animals
  ADD COLUMN gps_validity_code integer;
```

Now create a table to store the validity codes, create the external key and insert the admitted values:

```
CREATE TABLE lu_tables.lu_gps_validity(
  gps_validity_code integer,
  gps_validity_description character varying,
  CONSTRAINT lu_gps_validity_pkey
    PRIMARY KEY (gps_validity_code));
COMMENT ON TABLE lu_tables.lu_gps_validity
IS 'Look up table for GPS positions validity codes.';

ALTER TABLE main.gps_data_animals
  ADD CONSTRAINT animals_lu_gps_validity
  FOREIGN KEY (gps_validity_code)
  REFERENCES lu_tables.lu_gps_validity (gps_validity_code)
  MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION;

INSERT INTO lu_tables.lu_gps_validity
  VALUES (0, 'Position with no coordinate');
INSERT INTO lu_tables.lu_gps_validity
  VALUES (1, 'Valid Position');
INSERT INTO lu_tables.lu_gps_validity
  VALUES (2, 'Position with a low degree of reliability');
INSERT INTO lu_tables.lu_gps_validity
  VALUES (11, 'Position wrong: out of the study area');
INSERT INTO lu_tables.lu_gps_validity
  VALUES (12, 'Position wrong: impossible spike');
INSERT INTO lu_tables.lu_gps_validity
  VALUES (13, 'Position wrong: impossible place (e.g. lake or sea)');
INSERT INTO lu_tables.lu_gps_validity
  VALUES (21, 'Position wrong: duplicated timestamp');
```

Some errors are already contained in the five GPS data sets previously loaded into the database, but a new (fake) data set can be imported to verify a wider range of errors. To do this, insert a new animal, a new GPS sensor, a new deployment record and finally import the data from the .csv file provided in the test data set.

Insert a new animal, called '*test*':

```
INSERT INTO main.animals
  (animals_id, animals_code, name, sex, age_class_code, species_code, note)
  VALUES (6, 'test', 'test-ina', 'm', 3, 1, 'This is a fake animal, used to
  test outliers detection processes.');
```

Insert a new sensor, called '*GSM_test*':

```
INSERT INTO main.gps_sensors
  (gps_sensors_id, gps_sensors_code, purchase_date, frequency, vendor, model,
  sim)
  VALUES (6, 'GSM_test', '2005-01-01', 1000, 'TNT', 'top', '+391441414');
```

Insert the time interval of the deployment of the test sensor on the test animal:

```
INSERT INTO main.gps_sensors_animals
  (animals_id, gps_sensors_id, start_time, end_time, notes)
  VALUES (6, 6, '2005-04-04 08:00:00+02', '2005-05-06 02:00:00+02', 'test
  deployment');
```

The last step is importing the data set from the .csv file:

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol, temp,
  easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM_test.csv'
    WITH (FORMAT csv, HEADER, DELIMITER ';');
```

Now you can proceed with outlier detection, having a large set of errors to hunt for. You can start by assuming that all the GPS positions are correct (value '1'):

```
UPDATE main.gps_data_animals
  SET gps_validity_code = 1;
```

Missing Records

You might have a missing record when the device was programmed to acquire the position but no information (not even the acquisition time) is recorded. In this case, you can use specific functions (see Chap. 9) to create 'virtual' records and, if needed, compute and interpolate values for the coordinates. The 'virtual' records should be created just in the analytical stage and not stored in the reference data set (table *gps_data_animals*).

Records with Missing Coordinates

When the GPS is unable to receive sufficient satellite signal, the record has no coordinates associated. The rate of GPS failure can vary substantially, mainly according to sensor quality, terrain morphology and vegetation cover. Missing coordinates cannot be managed as location bias, but have to be properly treated in the analytical stage depending on the specific objective, since they result in an

erroneous 'fix rate' (see Special Topic—how to deal with erroneous fix rates is beyond the scope of this chapter). Technically, they can be filtered from the data set, or an estimated value can be calculated by interpolating the previous and next GPS positions. This is a very important issue, since several analytical methods require regular time intervals. Note that with no longitude/latitude, the spatial attribute (i.e. the *geom* field) cannot be created, which makes it easy to identify this type of error. You can mark all the GPS positions with no coordinates with the code '0':

```
UPDATE main.gps_data_animals
  SET gps_validity_code = 0
  WHERE geom IS NULL;
```

Multiple Records with the Same Acquisition Time

In some (rare) cases, you might have a repeated acquisition time (from the same acquisition source). You can detect these errors by grouping your data set by animal and acquisition time and asking for multiple occurrences. Here is an example of an SQL query to get this result:

```
SELECT
  x.gps_data_animals_id, x.animals_id, x.acquisition_time
FROM
  main.gps_data_animals x,
  (SELECT animals_id, acquisition_time
  FROM main.gps_data_animals
  WHERE gps_validity_code = 1
  GROUP BY animals_id, acquisition_time
  HAVING count(animals_id) > 1) a
WHERE
  a.animals_id = x.animals_id AND
  a.acquisition_time = x.acquisition_time
ORDER BY
  x.animals_id, x.acquisition_time;
```

This query returns the id of the records with duplicated timestamps (having *count(animals_id)* > 1). In this case, it retrieves two records with the same acquisition time ('2005-05-04 22:01:24+00'):

```
gps_data_animals_id | animals_id |    acquisition_time
--------------------+------------+------------------------
              28177 |          6 | 2005-05-05 00:01:24+02
              28176 |          6 | 2005-05-05 00:01:24+02
```

At this point, the data manager has to decide what to do. You can keep one of the two (or more) GPS positions with repeated acquisition time, or tag both (all) as

unreliable. The first possibility would imply a detailed inspection of the locations at fault, in order to possibly identify (with no guarantee of success) which one is correct. On the other hand, the second case is more conservative and can be automated as the user does not have to take any decision that could lead to erroneous conclusions. As for the other type of errors, a specific *gps_validity_code* is suggested. Here is an example:

```
UPDATE main.gps_data_animals
  SET gps_validity_code = 21
  WHERE
    gps_data_animals_id in
      (SELECT x.gps_data_animals_id
       FROM
        main.gps_data_animals x,
        (SELECT animals_id, acquisition_time
         FROM main.gps_data_animals
         WHERE gps_validity_code = 1
         GROUP BY animals_id, acquisition_time
         HAVING count(animals_id) > 1) a
       WHERE
        a.animals_id = x.animals_id AND
        a.acquisition_time = x.acquisition_time);
```

If you rerun the above query to identify locations with the same timestamps, it will now return an empty output.

Records with Different Values When Acquired Using Different Acquisition Sources

It may happen that data are obtained from sensors through different data transfer processes. A typical example is data received in near real time through a GSM network and later downloaded directly via cable from the sensor when it is physically removed from the animal. If the information is different, it probably means that an error occurred during data transmission. In this case, it is necessary to define a hierarchy of reliability for the different sources (e.g. data obtained via cable download are better than those obtained via the GSM network). This information should be stored when data are imported into the database into *gps_data* table. Then, when valid data are to be identified, the 'best' code should be selected, paying attention to properly synchronise *gps_data* and *gps_data _animals.* Which specific tools will be used to manage different acquisition sources largely depends on the number of sensors, frequency of updates and desired level of automation of the process. No specific examples are provided here.

Records Erroneously Attributed to Animals

This situation usually occurs for the first and/or last GPS positions because the start and end date and time of the sensor deployment are not correct. The consequence is that the movements of the sensor before and after the deployment are attributed to the animal. It may be difficult to trap this error with automatic methods because incorrect records can be organised in spatial clusters with a (theoretically) meaningful pattern (the set of erroneous GPS positions has a high degree of spatial autocorrelation as it contains 'real' GPS positions of 'real' movements, although they are not animal's movements). It is important to stress that this kind of pattern, e.g. GPS positions repeated in a small area where the sensor is stored before the deployment (e.g. the researcher's office) and then a long movement to where the sensor is deployed on the animal, can closely resemble the sequence of GPS positions for animals just released in a new area.

To identify this type of error, the suggested approach is to visually explore the data set in a GIS desktop interface. Once you detect this situation, you should check the source of information on the date of capture and sensor deployment and, if needed, correct the information in the table *gps_sensors_animals* (this will automatically update the table *gps_data_animals*). In general, a visual exploration of your GPS data sets, using as representation both points and trajectories, is always useful to help identify unusual spatial patterns. For this kind of error, no *gps_validity_code* are used because, once detected, they are automatically excluded from the table *gps_data_animals*.

The best method to avoid this type of error is to get accurate and complete information about the deployment of the sensors on the animals, for example, verifying not just the starting and ending date, but also the time of the day and time zone.

Special attention must be paid to the end of the deployment. For active deployments, no end is defined. In this case, the procedure can make use of the function *now()* to define a dynamic upper limit when checking the timestamp of recorded locations (i.e. the record is not valid if $acquisition\_time > now()$).

The next types of error can all be connected to GPS sensor malfunctioning or poor performance, leading to biased locations with low accuracy, or a true 'outlier', i.e. coordinates that are distant or very distant from the 'true location'.

Records Located Outside the Study Area

When the error of coordinates is due to reasons not related to general GPS accuracy (which will almost always be within a few dozen metres), the incorrect positions are often quite evident as they are usually very far from the others (a typical example is the inversion of longitude and latitude). At the same time, this error is random, so erroneous GPS positions are hardly grouped in spatial clusters.

When a study area has defined limits (e.g. fencing or natural boundaries), the simplest way to identify incorrect GPS positions is to run a query that looks for those that are located outside these boundaries (optionally, with an extra buffer area). Though animals have no constraints to their movements, they are still limited to a specific area (e.g. an island), so you can delineate a very large boundary so that at least GPS positions very far outside this area are captured. In this case, it is better to be conservative and enlarge the study area as much as possible to exclude all the valid GPS positions. Other, more fine-tuned methods can be used at a later stage to detect the remaining erroneous GPS positions. This approach has the risk of tagging correct locations if the boundaries are not properly set, as the criteria are very subjective. It is important to note that erroneous locations will be identified in any case as impossible movements (see next sections). This step can be useful in cases where you don't have access to movement properties (e.g. VHF data with only one location a week). Another element to keep in mind, especially in the case of automatic procedures to be run in real time on the data flow, is that very complex geometries (e.g. a coastline drawn at high spatial resolution) can slow down the intersection queries. In this case, you can exploit the power of spatial indexes and/or simplify your geometry, which can be done using the PostGIS commands *ST_Simplify*[1] and *ST_SimplifyPreserveTopology*[2]. Here is an example of an SQL query that detects outliers outside the boundaries of the *study_area* layer, returning the IDs of outlying records:

```
SELECT
  gps_data_animals_id
FROM
  main.gps_data_animals
LEFT JOIN
  env_data.study_area
ON
  ST_Intersects(gps_data_animals.geom, study_area.geom)
WHERE
  study_area IS NULL AND
  gps_data_animals.geom IS NOT NULL;
```

There result is the list of the six GPS positions that fall outside the study area:

```
 gps_data_animals_id
---------------------
               15810
               27945
               28094
               28111
               20540
               23030
```

[1] http://www.postgis.org/docs/ST_Simplify.html.

[2] http://www.postgis.org/docs/ST_SimplifyPreserveTopology.html.

The same query could be made using *ST_Disjoint*, i.e. the opposite of *ST_Intersects* (note, however, that the former does not work on multiple polygons). Here is an example where a small buffer (*ST_Buffer*) is added (using *Common Table Expressions*[3]):

```
WITH area_buffer_simplified AS
  (SELECT
    ST_Simplify(
      ST_Buffer(study_area.geom, 0.1), 0.1) AS geom
   FROM
    env_data.study_area)
SELECT
  animals_id, gps_data_animals_id
FROM
  main.gps_data_animals
WHERE
  ST_Disjoint(
    gps_data_animals.geom, (SELECT geom FROM area_buffer_simplified));
```

The use of the syntax with *WITH* is optional, but in some cases can be a useful way to simplify your queries, and it might be interesting for you to know how it works.

In this case, just five outliers are detected because one of the previous six is very close to the boundaries of the study area:

```
animals_id | gps_data_animals_id
-----------+--------------------
         3 |               15810
         6 |               27945
         6 |               28111
         1 |               20540
         5 |               23030
```

This GPS position deserves a more accurate analysis to determine whether it is really an outlier. Now tag the other five GPS positions as erroneous (validity code '11', i.e. 'Position wrong: out of the study area'):

```
UPDATE main.gps_data_animals
  SET gps_validity_code = 11
  WHERE
    gps_data_animals_id in
    (SELECT gps_data_animals_id
     FROM main.gps_data_animals, env_data.study_area
     WHERE ST_Disjoint(
       gps_data_animals.geom,
       ST_Simplify(ST_Buffer(study_area.geom, 0.1), 0.1)));
```

[3] http://www.postgresql.org/docs/9.2/static/queries-with.html.

Using a simpler approach, another quick way to detect these errors is to order GPS positions according to their longitude and latitude coordinates. The outliers are immediately visible as their values are completely different from the others and they pop up at the beginning of the list. An example of this kind of query is:

```
SELECT
  gps_data_animals_id, ST_X(geom)
FROM
  main.gps_data_animals
WHERE
  geom IS NOT NULL
ORDER BY
  ST_X(geom)
LIMIT 10;
```

The resulting data set is limited to ten records, as just a few GPS positions are expected to be affected by this type of error. From the result of the query, it is clear that the first two locations are outliers, while the third is a strong candidate:

```
 gps_data_animals_id |    st_x
---------------------+------------
               15810 |  5.0300699
               23030 | 10.7061637
               27948 | 10.9506126
               17836 | 10.9872122
               17835 | 10.9875451
               17837 | 10.9876942
               17609 | 10.9884574
               18098 | 10.9898182
               18020 | 10.9899461
               20154 | 10.9900441
```

The same query can then be repeated in reverse order, and then doing the same for latitude:

```
SELECT gps_data_animals_id, ST_X(geom)
FROM main.gps_data_animals
WHERE geom IS NOT NULL
ORDER BY ST_X(geom)
DESC LIMIT 10;

SELECT gps_data_animals_id, ST_Y(geom)
FROM main.gps_data_animals
WHERE geom IS NOT NULL
ORDER BY ST_Y(geom)
LIMIT 10;

SELECT gps_data_animals_id, ST_Y(geom)
FROM main.gps_data_animals
WHERE geom IS NOT NULL
ORDER BY ST_Y(geom) DESC
LIMIT 10;
```

Records Located in Impossible Places

When there are areas not accessible to animals because of physical constraints (e.g. fencing, natural barriers) or environments not compatible with the studied species (lakes and sea, or land, according to the species), you can detect GPS positions that are located in those areas where it is impossible for the animal to be. Therefore, the decision whether or not to mark the locations as incorrect is based on ecological assumptions (i.e. non-habitat). In this example, you mark, using validity code '13', all the GPS positions that fall inside a water body according to Corine land cover layer (Corine codes '40', '41', '42', '43' and '44'):

```
UPDATE main.gps_data_animals
  SET gps_validity_code = 13
  FROM
    env_data.corine_land_cover
  WHERE
    ST_Intersects(
      corine_land_cover.rast,
      ST_Transform(gps_data_animals.geom, 3035)) AND
    ST_Value(
      corine_land_cover.rast,
      ST_Transform(gps_data_animals.geom, 3035))
      in (40,41,42,43,44) AND
    gps_validity_code = 1 AND
    ST_Value(
      corine_land_cover.rast,
      ST_Transform(gps_data_animals.geom, 3035)) != 'NaN';
```

For this kind of control, you must also consider that the result depends on the accuracy of the land cover layer and of the GPS positions. Thus, at a minimum, a further visual check in a GIS environment is recommended.

Records that Would Imply Impossible Movements

To detect records with incorrect coordinates that cannot be identified using clear boundaries, such as the study area or land cover type, a more sophisticated outlier filtering procedure must be applied. To do so, some kind of assumption about the animals' movement model has to be made, for example, a speed limit. It is important to remember that animal movements can be modelled in different ways at different temporal scales: an average speed that is impossible over a period of 4 h could be perfectly feasible for movements in a shorter time (e.g. 5 minutes). Which algorithm to apply depends largely on the species and the environment in which the animal is moving and the duty cycle of the tag. In general, PostgreSQL window functions can help.

Special Topic: **PostgreSQL window functions**

A window function[4] performs a calculation across a set of rows that are somehow related to the current row. This is similar to an aggregate function, but unlike regular aggregate functions, window functions do not group rows into a single output row, hence they are still able to access more than just the current row of the query result. In particular, it enables you to access previous and next rows (according to a user-defined ordering criteria) while calculating values for the current row. This is very useful, as a tracking data set has a predetermined temporal order, where many properties (e.g. geometric parameters of the trajectory, such as turning angle and speed) involve a sequence of GPS positions. It is important to remember that the order of records in a database is irrelevant. The ordering criteria must be set in the query that retrieves data.

In the next example, you will make use of window functions to convert the series of locations into steps (i.e. the straight-line segment connecting two successive locations), and compute geometric characteristics of each step: the time interval, the step length, and the speed during the step as the ratio of the previous two. It is important to note that while a step is the movement between two points, in many cases, its attributes are associated with the starting or the ending point. In this book, we use the ending point as reference. In some software, particularly the adehabitat[5] package for *R* (see Chap. 10), the step is associated with the starting point. If needed, the queries and functions presented in this book can be modified to follow this convention.

```
SELECT
  animals_id AS id,
  acquisition_time,
  LEAD(acquisition_time,-1)
    OVER (
      PARTITION BY animals_id
      ORDER BY acquisition_time) AS acquisition_time_1,
  (EXTRACT(epoch FROM acquisition_time) -
  LEAD(EXTRACT(epoch FROM acquisition_time), -1)
    OVER (
      PARTITION BY animals_id
      ORDER BY acquisition_time))::integer AS deltat,
  (ST_Distance_Spheroid(
    geom,
    LEAD(geom, -1)
      OVER (
        PARTITION BY animals_id
        ORDER BY acquisition_time),
```

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

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

```
    'SPHEROID["WGS 84",6378137,298.257223563]'))::integer AS dist,
  (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))+1)*60*60)::numeric(8,2) AS speed
FROM main.gps_data_animals
WHERE gps_validity_code = 1
LIMIT 10;
```

The result of this query is

id	*acquisition_time*	*acquisition_time_1*	*deltat*	*dist*	*speed*
1	*2005-10-18 22:00:54+02*				
1	*2005-10-19 02:01:23+02*	*2005-10-18 22:00:54+02*	*14429*	*97*	*24.15*
1	*2005-10-19 06:02:22+02*	*2005-10-19 02:01:23+02*	*14459*	*430*	*107.08*
1	*2005-10-19 10:03:08+02*	*2005-10-19 06:02:22+02*	*14446*	*218*	*54.40*
1	*2005-10-20 22:00:53+02*	*2005-10-19 10:03:08+02*	*129465*	*510*	*14.17*
1	*2005-10-21 02:00:48+02*	*2005-10-20 22:00:53+02*	*14395*	*97*	*24.22*
1	*2005-10-21 06:00:53+02*	*2005-10-21 02:00:48+02*	*14405*	*69*	*17.26*
1	*2005-10-21 10:01:42+02*	*2005-10-21 06:00:53+02*	*14449*	*478*	*119.20*
1	*2005-10-21 18:01:16+02*	*2005-10-21 10:01:42+02*	*28774*	*150*	*18.77*
1	*2005-10-21 22:01:23+02*	*2005-10-21 18:01:16+02*	*14407*	*688*	*172.02*

As a demonstration of a possible approach to detecting 'impossible movements', here is an adapted function that implements the algorithm presented in Bjorneraas et al. (2010). In the first step, you compute the distance from each GPS position to the average of the previous and next ten GPS positions, and extract records that have values bigger than a defined threshold (in this case, arbitrarily set to 10,000 m):

```
SELECT gps_data_animals_id
FROM
  (SELECT
    gps_data_animals_id,
    ST_Distance_Spheroid(geom,
      ST_setsrid(ST_makepoint(
        avg(ST_X(geom))
          OVER (
```

```
            PARTITION BY animals_id
            ORDER BY acquisition_time rows
              BETWEEN 10 PRECEDING and 10 FOLLOWING),
        avg(ST_Y(geom))
          OVER (
            PARTITION BY animals_id
            ORDER BY acquisition_time rows
              BETWEEN 10 PRECEDING and 10 FOLLOWING)), 4326),'SPHEROID["WGS
              84", 6378137,298.257223563]') AS dist_to_avg
   FROM
     main.gps_data_animals
   WHERE
     gps_validity_code = 1) a
WHERE
  dist_to_avg > 10000;
```

The result is the list of IDs of all the GPS positions that match the defined conditions (and thus can be considered outliers). In this case, just one record is returned:

```
gps_data_animals_id
--------------------
              27948
```

This code can be improved in many ways. For example, it is possible to consider the median instead of the average. It is also possible to take into consideration that the first and last ten GPS positions have incomplete windows of 10 + 10 GPS positions. Moreover, this method works fine for GPS positions at regular time intervals, but in the case of a change in acquisition schedule might lead to unexpected results. In these cases, you should create a query with a temporal window instead of a fixed number of GPS positions.

In the second step, the angle and speed based on the previous and next GPS position are calculated (both the previous and next location are used to determine whether the location under consideration shows a spike in speed or turning angle), and then GPS positions below the defined thresholds (in this case, arbitrarily set as cosine of the relative angle <-0.99 and speed >2,500 m per hour) are extracted:

```
SELECT
  gps_data_animals_id
FROM
  (SELECT gps_data_animals_id,
  ST_Distance_Spheroid(
    geom,
    LAG(geom, 1) OVER (PARTITION BY animals_id ORDER BY acquisition_time),
    'SPHEROID["WGS 84",6378137,298.257223563]') /
    (EXTRACT(epoch FROM acquisition_time) - EXTRACT (epoch FROM
    (lag(acquisition_time, 1) OVER (PARTITION BY animals_id ORDER BY
```

```
    acquisition_time))))*3600 AS speed_FROM,
  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) + EXTRACT (epoch FROM
    (lead(acquisition_time, 1) OVER (PARTITION BY animals_id ORDER BY
    acquisition_time))))*3600 AS speed_to,
  cos(ST_Azimuth((
    LAG(geom, 1) OVER (PARTITION BY animals_id ORDER BY
    acquisition_time))::geography,
    geom::geography) -
  ST_Azimuth(
    geom::geography,
    (LEAD(geom, 1) OVER (PARTITION BY animals_id ORDER BY
    acquisition_time))::geography)) AS rel_angle
  FROM main.gps_data_animals
  WHERE gps_validity_code = 1) a
WHERE
  rel_angle < -0.99 AND
  speed_from > 2500 AND
  speed_to > 2500;
```

The result returns the list of IDs of all the GPS positions that match the defined conditions. The same record detected in the previous query is returned. These examples can be used as templates to create other filtering procedures based on the temporal sequence of GPS positions and the users' defined movement constraints.

It is important to remember that this kind of method is based on the analysis of the sequence of GPS positions, and therefore results might change when new GPS positions are uploaded. Moreover, it is not possible to run them in real time because the calculation requires a subsequent GPS position. The result is that they have to be run in a specific procedure unlinked with the (near) real-time import procedure.

Now you run this process on your data sets to mark the detected outliers (validity code '12'):

```
UPDATE
  main.gps_data_animals
SET
  gps_validity_code = 12
WHERE
  gps_data_animals_id in
    (SELECT gps_data_animals_id
    FROM
      (SELECT
```

```
      gps_data_animals_id,
      ST_Distance_Spheroid(geom, lag(geom, 1) OVER (PARTITION BY animals_id
      ORDER BY acquisition_time), 'SPHEROID["WGS 84",6378137,298.257223563]') /
      (EXTRACT(epoch FROM acquisition_time) - EXTRACT (epoch FROM
      (lag(acquisition_time, 1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))))*3600 AS speed_from,
      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) + EXTRACT (epoch FROM
      (lead(acquisition_time, 1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))))*3600 AS speed_to,
      cos(ST_Azimuth((lag(geom, 1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))::geography, geom::geography) - ST_Azimuth(geom::geography,
      (lead(geom, 1) OVER (PARTITION BY animals_id ORDER BY
      acquisition_time))::geography)) AS rel_angle
    FROM main.gps_data_animals
    WHERE gps_validity_code = 1) a
  WHERE
    rel_angle < -0.99 AND
    speed_from > 2500 AND
    speed_to > 2500);
```

Records that Would Imply Improbable Movements

This is similar to the previous type of error, but in this case, the assumption made in the animals' movement model cannot completely exclude that the GPS position is correct (e.g. same methods as before, but with reduced thresholds: cosine of the relative angle <-0.98 and speed >300 m per hour). These records should be kept as valid but marked with a specific validity code that can permit users to exclude them for analysis as appropriate.

```
UPDATE
  main.gps_data_animals
SET
  gps_validity_code = 2
WHERE
  gps_data_animals_id IN
    (SELECT gps_data_animals_id
    FROM
      (SELECT
        gps_data_animals_id,
        ST_Distance_Spheroid(geom, lag(geom, 1) OVER (PARTITION BY animals_id
        ORDER BY acquisition_time), 'SPHEROID["WGS 84",6378137,298.257223563]') /
        (EXTRACT(epoch FROM acquisition_time) - EXTRACT (epoch FROM
        (lag(acquisition_time, 1) OVER (PARTITION BY animals_id ORDER BY
        acquisition_time))))*3600 AS speed_FROM,
        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) + EXTRACT (epoch FROM
      (lead(acquisition_time, 1) OVER (PARTITION BY animals_id ORDER BY
```

```
    acquisition_time))))*3600 AS speed_to,
    cos(ST_Azimuth((lag(geom, 1) OVER (PARTITION BY animals_id ORDER BY
    acquisition_time))::geography, geom::geography) - ST_Azimuth(geom
    ::geography,(lead(geom, 1) OVER (PARTITION BY animals_id ORDER BY
    acquisition_time))::geography)) AS rel_angle
   FROM main.gps_data_animals
   WHERE gps_validity_code = 1) a
WHERE
  rel_angle < -0.98 AND
  speed_from > 300 AND
  speed_to > 300);
```

The marked GPS positions should then be inspected visually to decide if they are valid with a direct expert evaluation.

Update of Spatial Views to Exclude Erroneous Locations

As a consequence of the outlier tagging approach illustrated in these pages, views based on the GPS positions data set should exclude the incorrect points, adding a *gps_validity_code = 1* criteria (corresponding to GPS positions with no errors and valid geometry) in their *WHERE* conditions. You can do this for *analysis.view_convex_hulls*:

```
CREATE OR REPLACE VIEW analysis.view_convex_hulls AS
SELECT
  gps_data_animals.animals_id,
  ST_ConvexHull(ST_Collect(gps_data_animals.geom))::geometry(Polygon,4326)
  AS geom
FROM
  main.gps_data_animals
WHERE
  gps_data_animals.gps_validity_code = 1
GROUP BY
  gps_data_animals.animals_id
ORDER BY
  gps_data_animals.animals_id;
```

You do the same for *analysis.view_gps_locations*:

```
CREATE OR REPLACE VIEW analysis.view_gps_locations AS
SELECT
  gps_data_animals.gps_data_animals_id,
  gps_data_animals.animals_id,
  animals.name,
```

```
    timezone('UTC'::text, gps_data_animals.acquisition_time) AS time_utc,
    animals.sex,
    lu_age_class.age_class_description,
    lu_species.species_description,
    gps_data_animals.geom
  FROM
    main.gps_data_animals,
    main.animals,
    lu_tables.lu_age_class,
    lu_tables.lu_species
  WHERE
    gps_data_animals.animals_id = animals.animals_id AND
    animals.age_class_code = lu_age_class.age_class_code AND
    animals.species_code = lu_species.species_code AND
    gps_data_animals.gps_validity_code = 1;
```

Now repeat the same operation for *analysis.view_trajectories*:

```
CREATE OR REPLACE VIEW analysis.view_trajectories AS
SELECT
  sel_subquery.animals_id,
  st_MakeLine(sel_subquery.geom)::geometry(LineString,4326) AS geom
FROM
  (SELECT
    gps_data_animals.animals_id,
    gps_data_animals.geom,
    gps_data_animals.acquisition_time
  FROM main.gps_data_animals
  WHERE gps_data_animals.gps_validity_code = 1
  ORDER BY gps_data_animals.animals_id, gps_data_animals.acquisition_time)
  sel_subquery
GROUP BY sel_subquery.animals_id;
```

If you visualise these layers in a GIS desktop, you can verify that outliers are now excluded. An example for the MCP is illustrated in Fig. 8.2, which can be compared with Fig. 5.4.

Update Import Procedure with Detection of Erroneous Positions

Some of the operations to filter outliers can be integrated into the procedure that automatically uploads GPS positions into the table *gps_data_animals*. In this example, you redefine the *tools.new_gps_data_animals()* function to tag GPS positions with no coordinates (*gps_validity_code = 0*) and GPS positions outside of the study area (*gps_validity_code = 11*) as soon as they are imported into the database. All the others are set as valid (*gps_validity_code = 1*).

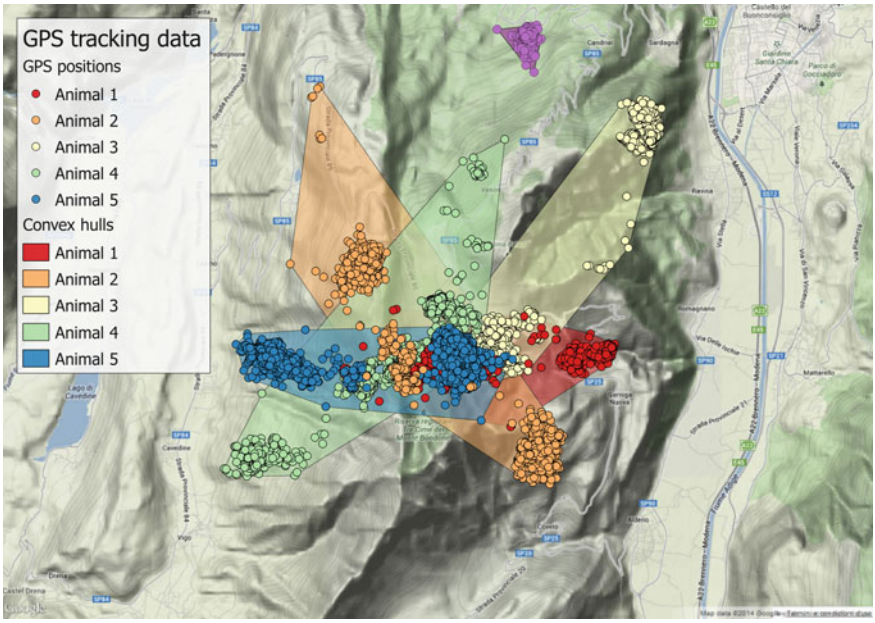

Fig. 8.2 Minimum convex polygons without outliers

```
CREATE OR REPLACE FUNCTION tools.new_gps_data_animals()
RETURNS trigger AS
$BODY$
DECLARE
thegeom geometry;
BEGIN
IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
  thegeom = ST_setsrid(ST_MakePoint(NEW.longitude, NEW.latitude), 4326);
  NEW.geom =thegeom;
  NEW.gps_validity_code = 1;
    IF NOT EXISTS (SELECT study_area FROM env_data.study_area WHERE
ST_intersects(ST_Buffer(thegeom,0.1), study_area.geom)) THEN
      NEW.gps_validity_code = 11;
    END IF;
  NEW.pro_com = (SELECT pro_com::integer FROM env_data.adm_boundaries WHERE
ST_intersects(geom,thegeom));
  NEW.corine_land_cover_code = (SELECT ST_Value(rast, ST_Transform(thegeom,
  3035)) FROM env_data.corine_land_cover WHERE
ST_Intersects(ST_Transform(thegeom,3035), rast));
  NEW.altitude_srtm = (SELECT ST_Value(rast,thegeom) FROM env_data.srtm_dem
```

```
WHERE ST_intersects(thegeom, rast));
  NEW.station_id = (SELECT station_id::integer FROM env_data.meteo_stations
ORDER BY ST_Distance_Spheroid(thegeom, geom, 'SPHEROID["WGS
  84",6378137,298.257223563]') LIMIT 1);
  NEW.roads_dist = (SELECT ST_Distance(thegeom::geography,
  geom::geography)::integer FROM env_data.roads ORDER BY
ST_Distance(thegeom::geography, geom::geography) LIMIT 1);
  NEW.ndvi_modis = (SELECT ST_Value(rast, thegeom)FROM env_data_ts.ndvi_modis
WHERE ST_Intersects(thegeom, rast)
AND EXTRACT(year FROM acquisition_date) = EXTRACT(year FROM
  NEW.acquisition_time)
AND EXTRACT(month FROM acquisition_date) = EXTRACT(month FROM
  NEW.acquisition_time)
AND EXTRACT(day FROM acquisition_date) = CASE
WHEN EXTRACT(day FROM NEW.acquisition_time) < 11 THEN 1
WHEN EXTRACT(day FROM NEW.acquisition_time) < 21 THEN 11
ELSE 21
END);
ELSE
NEW.gps_validity_code = 0;
END IF;
RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.new_gps_data_animals()
IS 'When called by the trigger insert_gps_locations (raised whenever a new GPS
position is uploaded into gps_data_animals) this function gets the new
longitude and latitude values and sets the field geom accordingly, computing a
set of derived environmental information calculated intersection the GPS
position with the environmental ancillary layers. GPS positions outside the
study area are tagged as outliers.';
```

You can test the results by reloading the GPS positions into *gps_data_animals* (for example, modifying the *gps_sensors_animals* table). If you do so, do not forget to rerun the tool to detect GPS positions in water, impossible spikes, and duplicated acquisition time, as they are not integrated in the automated upload procedure.

References

Bjorneraas K, van Moorter B, Rolandsen CM, Herfindal I (2010) Screening GPS location data for errors using animal movement characteristics. J Wild Manage 74:1361–1366. doi:10.1111/j.1937-2817.2010.tb01258.x

Frair JL, Fieberg J, Hebblewhite M, Cagnacci F, DeCesare NJ, Pedrotti L (2010) Resolving issues of imprecise and habitat-biased locations in ecological analyses using GPS telemetry data. Philos Trans R Soc B 365:2187–2200. doi:10.1098/rstb.2010.0084

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
```

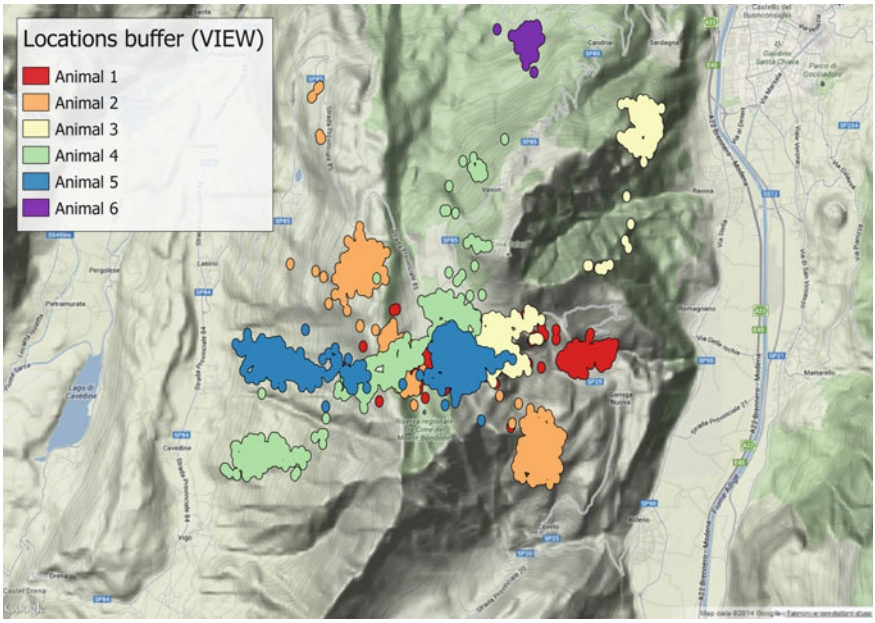

Fig. 9.1 Layer with dissolved buffers around GPS locations

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.

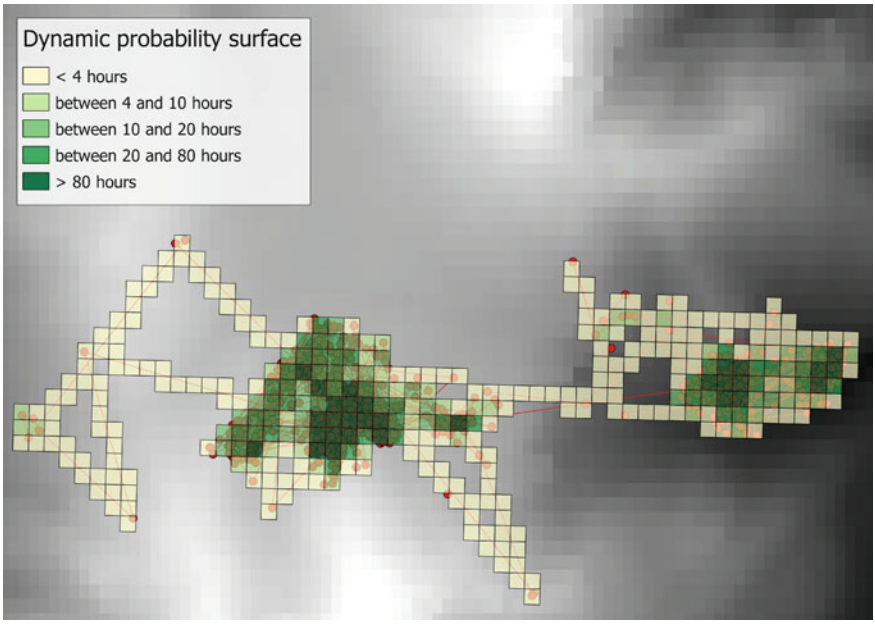

Fig. 9.2 Probability surface with *analysis.view_probability_grid_traj* (a DEM is visualised in the background)

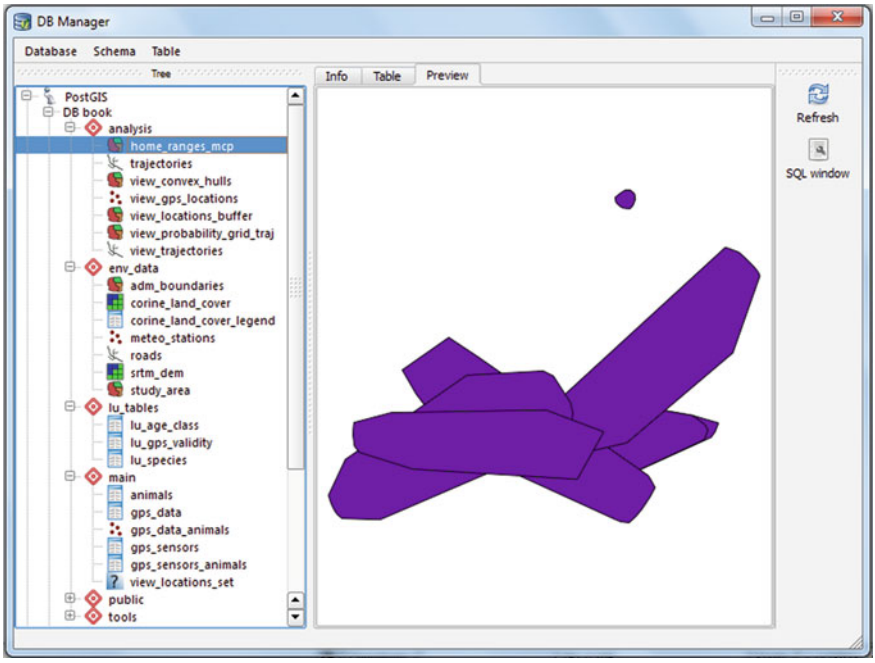

Fig. 9.3 Spatial content of the database as seen from DB Manager in QGIS

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

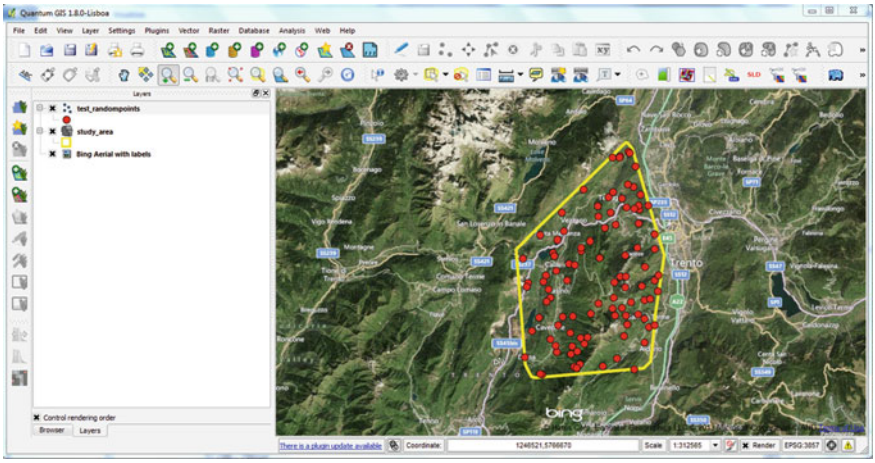

Fig. 9.4 Random points generated in a polygon

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

Chapter 10
From Data Management to Advanced Analytical Approaches: Connecting R to the Database

Bram Van Moorter

Abstract The previous chapters explored the wide set of tools that PostgreSQL and PostGIS offer to manage tracking data. In this chapter, you will expand database functionalities with those provided by dedicated software for statistical computing and graphics: the R programming language and software environment. Specifically, you will use the advanced graphics available through R and its libraries (especially '*adehabitat*') to perform exploratory analysis of animal tracking data. This data exploration is followed with two short ecological analyses. These ecological analyses are discussed in the context of two central concepts in animal space use: geographic versus environmental space, and the spatiotemporal scale of the research question. In the first analysis, you investigate the animal's home range, which is its use of geographic space. In the second, to explore an animal's use of environmental space we introduce briefly the study of both use and selection of environmental features by animals. For both demonstrations we consider explicitly the temporal scale of the study through the seasonal changes introduced by seasonal migration.

Introduction: From Data Management to Data Analysis

In previous chapters, you explored the wide set of tools that PostgreSQL and PostGIS offer to process and analyse tracking data. Nevertheless, a database is not specifically designed to perform advanced statistical analysis or to implement complex analytical algorithms, which are key elements to extract scientific knowledge from the data for both fundamental and applied research. In fact, these functionalities must be part of an information system that aims at a proper handling of wildlife tracking data. The possibility of a tighter integration of analytical functions with the database is

B. Van Moorter (✉)
Norwegian Institute for Nature Research (NINA), Høgskoleringen 9,
7034 Trondheim, Norway
e-mail: Bram.van.moorter@gmail.com

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_10,

particularly interesting because the availability of large amounts of information from the new generation sensors blurs the boundary between data analysis and data management. Tasks like outlier filtering, real-time detection of specific events (e.g. virtual fencing), or meta-analysis (analysis of results of a first analytical step, e.g. variation in home range size in the different months of a year) are clearly in the overlapping area between data analysis and management.

Background for the Analysis of Animal Space Use Data

Two questions need to be answered to move from an ecological question on animal space use to the analysis of those data: first, which is the relevant space (geographic versus environmental space), and second, which is the relevant spatio-temporal scale? Animals occupy a position in space at a given time *t*, which is called geographic space, and many ecological questions are related to this space: e.g. 'How large of an area is used by an animal during a year'? or 'How fast can an animal travel?'. Notably, the question on the area traversed by the animal has received much research interest, and this area is often called a 'home range'. The home range has been defined by Burt (1943) as 'the area traversed by the individual in its normal activities of food gathering, mating, and caring for young'. Many statistical approaches have been developed to use sets of location 'points' to estimate a home range area, from convex polygons to kernel density estimators (and many variants; for a review, see Kie et al. 2010).

On the other hand, by being in a certain geographic location, the animal encounters a set of environmental conditions, which are called environmental space, and questions related to the animal's ecological relationships are to be answered in this space: e.g. 'Which environmental characteristics does the animal prefer'? or 'How does human land use affect the animal's space use?'. These questions are the main topic of habitat selection studies. In general, in these studies, one compares the environmental conditions used by the animal to those available to the animal. An important challenge is to decide 'What environmental conditions were available to the animal'? This issue is tightly linked to the next question to be answered about scale.

The second important question for animal space use studies is about the relevant scale for the analysis. Small-scale studies can focus on the spatial behaviour of animals within a day or even an hour, whereas large-scale studies can look at space use over a year or even an animal's lifetime. The required scale of the study leads to certain demands on the data as well: a small-scale study requires high-resolution collection of precise locations, whereas a large-scale study will require a sufficient tracking duration to allow inferences over this long period. It seems obvious that any description of the area traversed by an individual must specify the time period over which the traversing occurred. Only for stable home ranges is it so that after a certain amount of time the size of the range no longer increases and the area becomes no longer time-dependent. Although often assumed, the stability

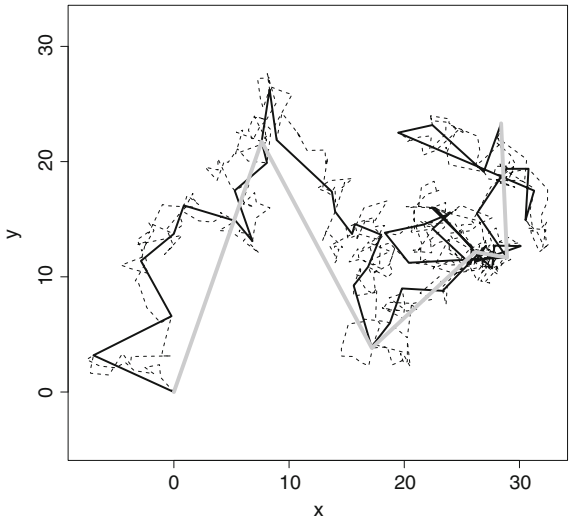

Fig. 10.1 This simulated trajectory clearly illustrates that reducing the number of acquired fixes within a time period can alter substantially the properties of the trajectory. The *dashed line* is the original trajectory, sampling with a ten times coarser resolution leads to a shorter trajectory in *black*, and a further reduction results in an even shorter trajectory in *gray*. Even though the total length of the trajectory decreases with a reduction in resolution, the distance between the consecutive points increases. It is thus very important for the researcher to be aware of such effects of sampling scale on trajectory characteristics

of a home range should be tested as often this is not the case. Also the distance travelled is very sensitive to the scale of the study, see Fig. 10.1.

For habitat selection studies—i.e. studies in environmental space—scale not only affects the sampling of the animal's space use, but even more important also the sampling of the environmental conditions available to the animal to choose from. Areas available to an animal in the course of several days may not be reachable within the time span of a few hours. This behavioural limitation has led several studies to consider availability specific to each used location (e.g. step-selection functions, Fortin et al. (2005)), instead of considering the same choice set available for all locations of a single individual (or even population). Thus, how the researcher defines the available choice set should be informed by the scale of the study.

The Tools: R and Adehabitat

R is an open source programming language and environment for statistical computing and graphics (http://www.r-project.org/). It is a popular choice for data analysis in academics, with its popularity for ecological research increasing rapidly. This popularity is not only the result of R being available for free, but also

due to its flexibility and extendibility. The large user base of R has developed an extensive suite of libraries to extend the basic functionalities provided in the R-base package. Before going into some of these really fantastic features, it is good to point out that R's flexibility comes at a cost: a rather steep-learning curve. R is not very tolerant to small human mistakes and demands of the first-time user some initial investment to understand its sometimes cryptic error messages. Fortunately, there are many resources out there to help the novice on her way (look on the R website for more options): online tutorials (e.g. 'R for Beginners' or 'Introduction to R'), books (a popular choice is Crawley's (2005) 'Statistics: An Introduction using R'), extensive searchable online help (e.g. http://www.rseek.org/), and the many statistics courses that include an introduction to R (search the Internet for a course nearby; some may even be offered online).

One of the main strengths of R is the availability of a large range of packages or libraries for specific applications; by 2013, more than 4,000 libraries were published on CRAN. There are libraries for advanced statistical analysis (e.g. '*lme4*' for mixed-effects models or '*mgcv*' for general additive models), advanced graphics (e.g. '*ggplot2*'), spatial analysis (e.g. '*sp*'), or database connections (e.g. '*RPostgreSQL*'). Many packages have been developed to allow the use of R as a general interface to interact with databases or GIS. Other packages have gone even further and increase the performance of R in fields for which it was not originally developed such as the handling of very large data sets or GIS. Hence, many different R users have come into existence: from users relying on specific software for specific tasks who use R exclusively for their statistical analysis and use different files to push data through their workflow, to the other extreme of users who use R to control a workflow in which R sometimes calls on external software to get the job done, but more often with the help of designated libraries, R gets the job done itself. Instead, the approach advocated in this book is not centred around software, but places the data in the centre of the workflow. The data are stored in a spatial database, and all software used interacts with this database. You have seen several examples in this book using different software (e.g. pgadmin or QGIS); in this chapter, you will use R as another option to perform some additional specific tasks with the data stored in the database.

For the analysis of animal tracking data, we often use functions from adehabitat (Calenge 2006), which today consists of '*adehabitatLT*' (for the analysis of trajectories), '*adehabitatHR*' (for home range estimation), '*adehabitatHS*' (for habitat-selection analysis), and '*adehabitatMA*' (for the management of raster maps). For the general management of spatial data, we rely on the '*sp*'- and '*rgdal*'-libraries, for the advanced management of raster data; the '*raster*'-library is becoming the standard. We recommend as a general introduction to the use of spatial data in R the book by Bivand et al. (2008). To install a package with a library in R, you use the *install.packages* command, as illustrated here for the *'adehabitat'*-libraries:

```
# comments in R start with a '#', and what follows will be ignored by R
# to install the adehabitat packages:
install.packages("adehabitatLT")
install.packages("adehabitatHR")
install.packages("adehabitatHS")
install.packages("adehabitatMA")
```

The different adehabitat packages come with extensive tutorials, which are accessible in R through

```
vignette("adehabitatLT")
vignette("adehabitatHR")
vignette("adehabitatHS")
vignette("adehabitatMA")
```

Before using the database to analyse your animal tracking data with R, you can use adehabitat to replicate Fig. 10.1:

```
library(adehabitatLT)
set.seed(0)  #this allows you to replicate
# the exact same 'random' numbers as we did for the figure
simdat <- simm.crw(c(1:501))[[1]]
plot(simdat$x, simdat$y, type = "l", lty = "dashed", xlab = "x", ylab = "y",
     asp = T)
lines(simdat$x[seq(1, 501, by = 10)], simdat$y[seq(1, 501, by = 10)], lwd = 2)
lines(simdat$x[seq(1, 501, by = 100)], simdat$y[seq(1, 501, by = 100)], lwd = 4,
      col = "grey")
```

The *simm.crw* function simulates a random walk. A random walk is a movement where the direction and the distance of each consecutive location are randomised; it is therefore also referred to as a 'drunkard's walk'. It is beyond the scope of this chapter to give an introduction to random walks (see e.g. Turchin 1998 for more on random walks to model movement of organisms).

You can find more information on a function with a '?' in front of the function; this will access its associated help pages with explanation and working examples of the function's uses:

```
`?`(simm.crw)
```

You can see in these help pages that there are several parameters that can be altered to make the random walk behave differently.

Connecting R to the Database

To use R for the analysis of the data stored and managed within the database, there are two approaches: first, connect from R to the database, and second, connect from the database to R with the Pl/R-interface. We will start by demonstrating the first approach and using it for some exercises. In the next chapter, you will see how this approach can be extended to connect from within the database to R with the PostgreSQL procedural language Pl/R (www.joeconway.com/plr/doc/).

First, to connect from R to the database, we make use of the '*RPostgreSQL*' library. It is possible with '*rgdal*' to read spatial features from PostGIS directly in R into '*sp*''s spatial classes. However, using '*rgdal*', it is no longer possible to perform SQL operations (such as SELECT) on these data. One solution could be to create in the database a temporary table with the selected spatial features and then use '*rgdal*' to read the spatial features into R. However, the performance of '*rgdal*' is considerably lower—it can be 100 times slower for certain operations—than for the database libraries, such as '*RPostgreSQL*'. Unfortunately, to date, there is no straightforward way for Windows users to read spatial data into R using SQL statements from a PostgreSQL-database. Thus, when you want to include an SQL statement, you will have to convert the data to non-spatial classes and then subsequently convert them back to spatial features in R. In the next chapter, we will discuss the pros and cons of the use of R within the database through Pl/R.

To connect to a PostgreSQL-database, we use the '*RPostgreSQL*' library. The driver is '*PostgreSQL*' for a PostgreSQL-database as yours. The connection requires information on the driver, database name, host, port, user, and password. Except from the driver, all other parameters may have to be adjusted for your own specific case. If you have the database on your own machine, then the host and port will likely be 'localhost' and 5432, respectively, as shown here. You can see the tables in the database with the *dbListTables* command:

```
library(RPostgreSQL)
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname="gps_tracking_db", host="localhost",
                 port="5432", user="postgres", password="********")
dbListTables(con)
## Loading required package:  DBI
```

```
##  [1] "lu_species"                "lu_age_class"
##  [3] "gps_data_animals"          "gps_sensors"
##  [5] "gps_data"                  "gps_sensors_animals"
##  [7] "spatial_ref_sys"           "meteo_stations"
##  [9] "study_area"                "roads"
## [11] "adm_boundaries"            "srtm_dem"
## [13] "corine_land_cover"         "corine_land_cover_legend"
## [15] "lu_gps_validity"           "ndvi_modis"
## [17] "trajectories"              "animals"
## [19] "home_ranges_mcp"           "test_randompoints"
## [21] "activity_sensors_animals" "activity_data"
## [23] "activity_sensors"          "activity_data_animals"
```

The following code retrieves the first five lines from the gps_data_animals table:

```
fetch(dbSendQuery(con, "SELECT * FROM main.gps_data_animals LIMIT 5;"), -1)

##   gps_data_animals_id gps_sensors_id animals_id    acquisition_time
## 1               28250              6          6 2005-04-08 10:01:24
## 2               28344              6          6 2005-04-18 14:00:47
## 3               28396              6          6 2005-04-27 06:01:54
## 4               26109              1          2 2005-08-01 21:00:30
## 5               26684              1          2 2005-10-15 20:32:17
##   longitude latitude    insert_timestamp    update_timestamp
## 1     10.95    45.97 2013-09-12 18:45:17 2013-10-10 16:08:23
## 2     11.04    46.06 2013-09-12 18:45:17 2013-10-10 16:08:44
## 3     11.10    46.07 2013-09-12 18:45:17 2013-10-10 16:08:44
## 4     11.03    46.03 2013-09-12 09:56:38 2013-10-10 16:08:44
## 5     11.02    46.02 2013-09-12 09:56:38 2013-10-10 16:08:44
##                                                 geom pro_com
## 1 0101000020E6100000C054D8B1B6E62540169C6626BDFB4640      NA
## 2 0101000020E6100000D9EBDD1FEF15264043812D65CF074740   22205
## 3 0101000020E610000099BDC7F4DF3226409FD9BFFC5F094740   22205
## 4 0101000020E6100000CA6E66F4A30D2640CA0E3B9D75034740   22101
## 5 0101000020E61000000C186E0A750A2640AF04F7A864024740   22101
##   corine_land_cover_code altitude_srtm station_id roads_dist ndvi_modis
## 1                     21           411          1        119         NA
## 2                     25           884          3       2167         NA
## 3                     25           357          3       1243         NA
## 4                     24          1681          5        372         NA
## 5                     24          1666          5       1173         NA
##   gps_validity_code
## 1                12
## 2                 2
## 3                 2
## 4                 2
## 5                 2
```

You can see that R did not understand the *geom* column correctly.

Now, you want to retrieve all the necessary information for your analyses of roe deer space use. You first send a query to the database:

```
rs <- dbSendQuery(con, "SELECT animals_id, acquisition_time,longitude, latitude,
                  ST_X(ST_Transform(geom, 32632)) as x32,
                  ST_Y(ST_Transform(geom, 32632)) as y32, roads_dist,
                  ndvi_modis, corine_land_cover_code, altitude_srtm
                  FROM main.gps_data_animals where gps_validity_code = 1;"  )
locs <- fetch(rs,-1)
dbClearResult(rs)
```

You then need to fetch those data (with the -1, you indicate that you want all data), and then 'clear' the result set. Virtually all spatial operations (such as projection) could also be done in R; however, it is faster and easier to have the database project the data to UTM32 (which has the SRID code 32632).

```
head(locs)

##   animals_id    acquisition_time longitude latitude    x32     y32
## 1          2 2005-03-21 01:03:06     11.07    45.99 660151 5095008
## 2          2 2005-03-21 13:02:19     11.07    46.00 660004 5095733
## 3          2 2005-03-21 21:01:49     11.07    46.00 659954 5095673
## 4          2 2005-03-21 05:01:45     11.07    45.99 660104 5095155
## 5          6 2005-04-05 20:02:48     11.06    46.07 659592 5103359
## 6          6 2005-04-06 04:01:46     11.06    46.07 659631 5103352
##   roads_dist ndvi_modis corine_land_cover_code altitude_srtm
## 1        682       5048                     23          1085
## 2       1139       4314                     25          1378
## 3       1214       4797                     25          1401
## 4        834       5048                     23          1201
## 5       1205       5793                     19           740
## 6       1168       5793                     19           740
```

The *head* function allows us to inspect the first lines (by default six) of a dataframe. You see that you have successfully imported your data into R.

For dates, you should always carefully inspect their time zone. Due to the different time zones in the world, it is easy to get errors and make mistakes in the treatment of dates:

```
head(locs$acquisition_time)

## [1] "2005-03-21 01:03:06 CET"  "2005-03-21 13:02:19 CET"
## [3] "2005-03-21 21:01:49 CET"  "2005-03-21 05:01:45 CET"
## [5] "2005-04-05 20:02:48 CEST" "2005-04-06 04:01:46 CEST"
```

The time zone is said to be CEST and CET (i.e. Central European Summer Time and Central European Winter Time), note that the time zone will depend upon the local settings of your computer. However, we know that the actual time zone of these data is UTC (i.e. Universal Time or Greenwich Mean Time).

Let us then inspect whether the issue is an automatic transformation of the time zone, or whether the time zone was not correctly imported. With the library '*lubridate*', you can easily access the hour or month of a *POSIXct*-object:

```
library(lubridate)
table(month(locs$acquisition_time), hour(locs$acquisition_time))

##
##       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
##   1   0 152   0   0  36 118   0  35   0 126  33   0   0 149   0   0  35
##   2   0 121   0   0  33  97   0  31   0  95  34   0   0 116   0   0  42
##   3   0 129  21   0   0 122  23   0   0 127  24   0   0 114  20   0   0
##   4   9   0 168   0  10   0 171   0  11   0 164   0  11   0 156   0  11
##   5   5   1 158   1   6   1 147   1   6   1 122   0   5   1 149   0   5
##   6   0   0 125   0   0   0 107   0   0   0 100   0   0   0 107   0   0
##   7   2   2 144   2   2   2 114   1   1   2 128   2   2   2 110   2   2
##   8  10   8 137   7   8   8 131   7   9   8 126   7   8   8 112   9   9
##   9  10  10 136   7  10   9 142   8   8   7 112   8   6   7  99   4   7
##   10  8  16 161   8   8  23 154   8   8  20 132   8   8  18 128   7   7
##   11  0 181   0   0   0 182   0   0   0 167   0   0   0 166   0   0   0
##   12  4 196   6   6   6 193   5   5   6 178   5   6   5 183   6   5   6
##
##       17  18  19  20  21  22  23
##   1  118   0  33   0 118  35   0
##   2   95   0  31   0 103  33   0
##   3  130  27   0   0 123  30   0
##   4    0 175   0  12   0 180   0
##   5    1 128   1   5   1 151   1
##   6    0 102   0   0   0 125   0
##   7    2 103   2   0   0 131   1
##   8    7 104   6   8   8 139   9
##   9   10 117   8   9   7 145  10
##   10  17 138   8   7  20 161   8
##   11 179   0   0   0 184   0   0
##   12 197   5   5   6 198   6   5
```

The table shows us that there is a clear change in the frequency of the daily hours between March–April and October–November, which indicates the presence of daylight saving time. You can therefore safely assume that the UTC time in the database was converted to CEST/CET time.

To prevent mistakes due to daylight saving time, it is much easier to work with UTC time (UTC does not have daylight saving). Thus, you have to convert the dates back to UTC time. With the aforementioned *'lubridate'* library, you can do this easily: The function *with_tz* allows you to convert the local back to the UTC zone:

```
locs$UTC_time <- with_tz(locs$acquisition_time, tz = "UTC")
head(locs$acquisition_time)

## [1] "2005-03-21 01:03:06 CET"  "2005-03-21 13:02:19 CET"
## [3] "2005-03-21 21:01:49 CET"  "2005-03-21 05:01:45 CET"
## [5] "2005-04-05 20:02:48 CEST" "2005-04-06 04:01:46 CEST"

head(locs$UTC_time)

## [1] "2005-03-21 00:03:06 UTC" "2005-03-21 12:02:19 UTC"
## [3] "2005-03-21 20:01:49 UTC" "2005-03-21 04:01:45 UTC"
## [5] "2005-04-05 18:02:48 UTC" "2005-04-06 02:01:46 UTC"
```

Indeed, the *UTC_time*-column now contains the time zone: 'UTC'. You can run the command '*table(month(locs$UTC_time), hour(locs$UTC_time))*' to verify that no obvious shift in sampling occurred in the data. From personal experience, we know that many mistakes happen with time zones and daylight saving time, and we therefore recommend that you use UTC and carefully inspect your dates and ensure that they were correctly imported into R.

Data Inspection and Exploration

Before you dive into the analysis to answer your ecological question, it is crucial to perform a preliminary inspection of the data to verify data properties and ensure the quality of your data. Several of the following functionalities that are implemented in R can also easily (and more quickly) be implemented into the database itself. The main strength of R, however, lies in its visualisation capabilities. The visualisation of different aspects of the data is one of the major tasks during an exploratory analysis.

The basic trajectory format in adehabitat is *ltraj*, which is a list used to store trajectories from different animals. For more details on the *ltraj*-format, you refer to the vignettes (remember: *vignette(adehabitatLT)*) and the help pages for the '*adehabitatLT*'-library. An *ltraj*-object requires projected coordinates, a date for each location, and an animal identifier:

```
library(adehabitatLT)
ltrj <- as.ltraj(locs[, c("x32", "y32")], locs$UTC_time, locs$animals_id)
class(ltrj)

## [1] "ltraj" "list"

class(ltrj[[1]])

## [1] "data.frame"
```

The class-function shows us that *ltraj* is an object from the class *ltraj* and *list*. Each element of an *ltraj*-object is a *data.frame* with the trajectory information for each burst of each animal. A burst is a more or less intense monitoring of the animal followed by a gap in the data. For instance, animals that are only tracked during the day and not during the night will have for each day period a burst of data. The automatic schedule used for the GPS tracking of the roe deer in your database did not contain any intentional gaps; we therefore consider all data from an animal as belonging to a single burst.

```
head(ltrj[[1]])
```

```
##           x       y                date      dx      dy   dist     dt
## 1549 658249 5097296 2005-10-18 20:00:54   29.89   92.04  96.77  14429
## 1720 658279 5097388 2005-10-19 00:01:23   57.87 -426.16 430.07  14459
## 1025 658337 5096962 2005-10-19 04:02:22   46.07 -213.36 218.27  14446
## 1402 658383 5096749 2005-10-19 08:03:08 -229.96  454.70 509.55 129465
## 1298 658153 5097204 2005-10-20 20:00:53  -88.91   38.42  96.86  14395
## 1839 658064 5097242 2005-10-21 00:00:48  -63.56  -27.02  69.06  14405
##         R2n abs.angle rel.angle
## 1549      0     1.257        NA
## 1720   9365    -1.436   -2.6927
## 1025 119334    -1.358    0.0777
## 1402 317634     2.039   -2.8860
## 1298  17847     2.734    0.6947
## 1839  37195    -2.740    0.8099
```

The *head* function shows us that the *data.frames* within an *ltraj* object have ten columns. The first three columns define the location of the animal: the x and y coordinate and its date. The following columns describe the step (or change in location) toward the next location: the change in the x and y coordinates, the distance, the time interval between both locations, the direction of the movement, and the change in movement direction. The *R2n* is the squared displacement from the start point (or the net squared displacement [NSD]); we will discuss this metric in more detail later (see Calenge et al. 2009, and Fig. 10.2 for more explanation on these movement metrics).

Note that the animal's identifier is not in the table. As all locations belong to the same animal, there is no need to provide this information here. To obtain the identifiers of all the *data.frames* in an *ltraj*, you use the id function:

```
id(ltrj)
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

Or, to obtain the id of only one animal,

```
id(ltrj[1])
```

```
## [1] "1"
```

The summary function gives some basic information on the *ltraj*-object:

```
(sumy <- summary(ltrj))
```

```
##   id burst nb.reloc NAs          date.begin            date.end
## 1  1     1     1647   0 2005-10-18 20:00:54 2006-10-29 12:00:49
## 2  2     2     2194   0 2005-03-20 16:03:14 2006-05-27 16:02:25
## 3  3     3     1826   0 2005-10-23 20:00:53 2006-10-28 12:01:18
## 4  4     4     2641   0 2005-10-21 20:00:47 2007-02-09 08:11:24
## 5  5     5     2695   0 2006-11-13 00:02:24 2008-03-15 08:01:37
## 6  6     6      278   0 2005-04-04 06:01:41 2005-05-05 23:01:47
```

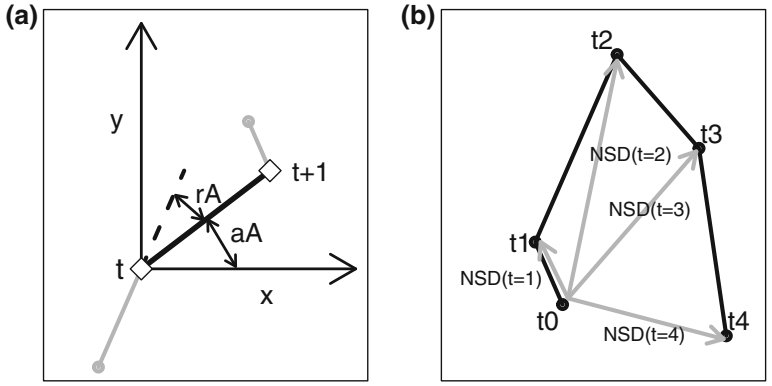

Fig. 10.2 The common trajectory characteristics stored in an *ltraj*. Panel **a** Shows the properties of one step from t to $t + 1$, and panel **b** the NSD of a series of locations ($t = 1$–5). The relative (rA) and absolute (aA) angles are also called the turning angle and direction of a step, respectively

Note that it marks 0 for missing values (i.e. NAs). However, this is not correct; you will see below how to tell R that there are missing observations in the data.

You see also that the total tracking duration is highly variable among individuals; notably Animal 6 was tracked for a much shorter time than the other animals. To ensure homogeneous data for the following analyses, you will only keep animals that have a complete year of data (i.e. number of days ≥365):

```
(duration <- difftime(sumy$date.end, sumy$date.begin, units = "days"))

## Time differences in days
## [1] 375.67 433.00 369.67 475.51 488.33  31.71
## attr(,"tzone")
## [1] "UTC"

ltrj <- ltrj[duration >= 365]
```

Moreover, for animals that were tracked for a longer period than one year, you remove locations in excess. Of course, if your ecological question is addressing space use during another period (e.g. spring) than you would want to keep all animals that provide this information, and Animal 6 may be retained for analysis, while removing all locations that are not required for this analysis.

```
ltrj <- cutltraj(ltrj,"difftime(date, date[1], units='days')>365")

summary (ltrj)

##   id  burst  nb.relocNAs         date.begin         date.end
## 1  1   1.01     1603   0 2005-10-18 20:00:54 2006-10-19 00:00:49
## 2  2 2.001     1894   0 2005-03-20 16:03:14 2006-03-20 20:02:06
## 3  3   3.01     1804   0 2005-10-23 20:00:53 2006-10-23 20:00:54
## 4  4 4.001     2009   0 2005-10-21 20:00:47 2006-10-21 20:00:54
## 5  5 5.001     1990   0 2006-11-13 00:02:24 2007-11-13 04:00:56
```

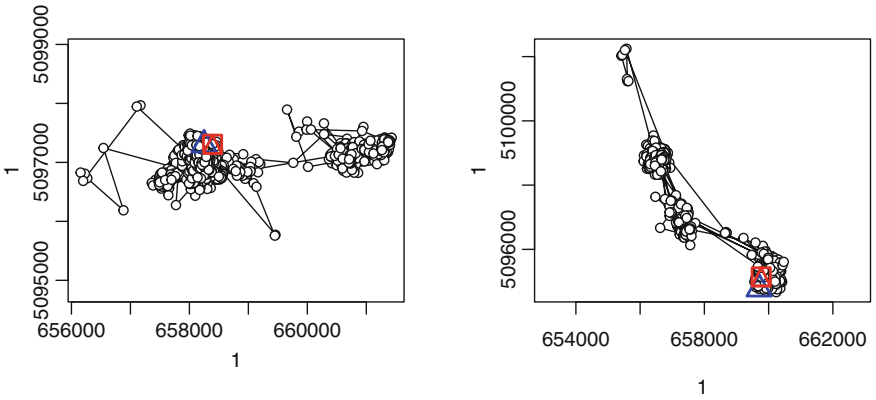

Fig. 10.3 Plot of the trajectories for the two first animals. You could have plotted all animals by simply running *plot(ltrj)*

Now, all animals are tracked for a whole year.

With the *plot* function, you can show the different trajectories (see the result in Fig. 10.3):

```
par(mfrow = c(1, 2))
plot(ltrj[1])
plot(ltrj[2])
```

The *plotltr* function allows us to show other characteristics than the spatial representation of the trajectory. For instance, it is very useful to get an overview of the sampling interval of the data. We discussed before how the sampling interval has a large effect on the patterns that you can observe in the data.

```
plotltr(ltrj[1], which = "dt/60/60", ylim = c(0, 24))
abline(h = seq(4, 24, by = 4), col = "grey")
```

Figure 10.4 shows that the time gap (dt) is not always constant between consecutive locations. Most often there is a gap of 4 h; however, there are several times that data are missing, and the gap is larger. Surprisingly, there are also locations where the gap is smaller than 4 h. We wrote a function to remove locations that are not part of a predefined sampling regime:

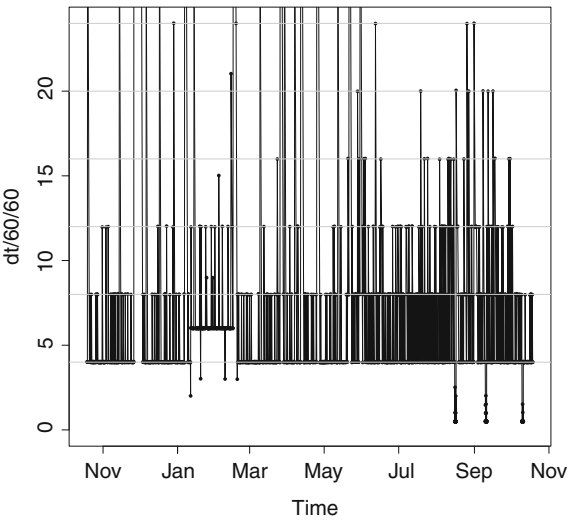

Fig. 10.4 The time interval between locations for animal 1

```
removeOutside <- function(x, date.ref, dt_hours = 1, tol_mins = 5) {
    require(adehabitatLT)
    x <- ld(x)
    tmp <- x$date + tol_mins * 60
    tmp_h <- as.POSIXlt(tmp)$hour
    tmp_m <- as.POSIXlt(tmp)$min
    hrs <- as.POSIXlt(date.ref)$hour
    hrs <- seq(hrs, 24, by = dt_hours)
    x <- x[tmp_h %in% hrs & tmp_m < (2 * tol_mins), ]
    x <- dl(x)
    return(x)
}
```

You now use this function on the *ltraj*; you specify a reference date at midnight, and the expected time lags in hours (dt_hours) is 4, and you set a tolerance of ±3 min (tol_mins):

```
ltrj <- removeOutside(ltrj, dt_hours = 4, tol_mins = 3,
                      date.ref=as.POSIXct("2005-01-01 00:00:00", tz="UTC"))
```

You can now inspect the time lag for each animal again:

```
plotltr(ltrj[1], which = "dt/60/60", ylim = c(0, 24))
abline(h = seq(4, 24, by = 4), col = "grey")
```

You see in Fig. 10.5 that there are no longer observations that deviate from the 4-hour schedule we programmed in our GPS sensors, i.e. all time lags between consecutive locations are a multiple of four. You still see gaps in the data, i.e. some

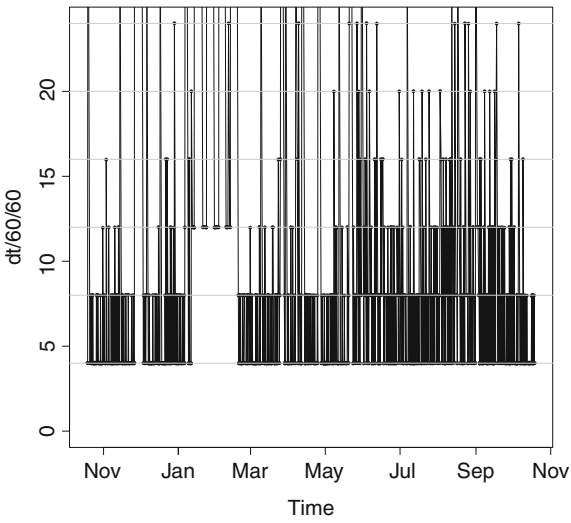

Fig. 10.5 The time interval between locations for animal 1 after the removal of locations outside the base sampling interval

gaps are larger than 4 h. Thus, there are missing values. The summary did not show the presence of missing data in the trajectory; you therefore have to specify the occurrence of missing locations.

The *setNA* function allows us to place the missing values into the trajectory at places where the GPS was expected to obtain a fix but failed to do so. You have to indicate a GPS schedule, which is in your case 4 h:

```
  ltrj <- setNA(ltrj, as.POSIXct("2004-01-01 00:00:00", tz="UTC"),
                dt=4*60*60)
  summary(ltrj)

##   id burst nb.reloc NAs          date.begin            date.end
## 1  1  1.01     2192 942 2005-10-18 20:00:54 2006-10-19 00:00:49
## 2  2 2.001     2189 758 2005-03-21 04:01:45 2006-03-20 20:02:06
## 3  3  3.01     2191 726 2005-10-23 20:00:53 2006-10-23 20:00:54
## 4  4 4.001     2191 492 2005-10-21 20:00:47 2006-10-21 20:00:54
## 5  5 5.001     2192 325 2006-11-13 00:02:24 2007-11-13 04:00:56
```

Indeed, now you see that the trajectories do contain a fair number of missing locations.

If locations are missing randomly, it will not bias the results of an analysis. However, when missing values occur in runs, this may affect your results. In adehabitat, there are two figures to inspect patterns in the missing data. The function *plotNAltraj* shows for a trajectory where the missing values occur and can be very instructive to show important gaps in the data:

```
da <- ltrj[[1]]$date
# the vector with dates allows us to zoom in on a range of dates in the plot
plotNAltraj(ltrj[1], xlim = c(da[1], da[500]))
```

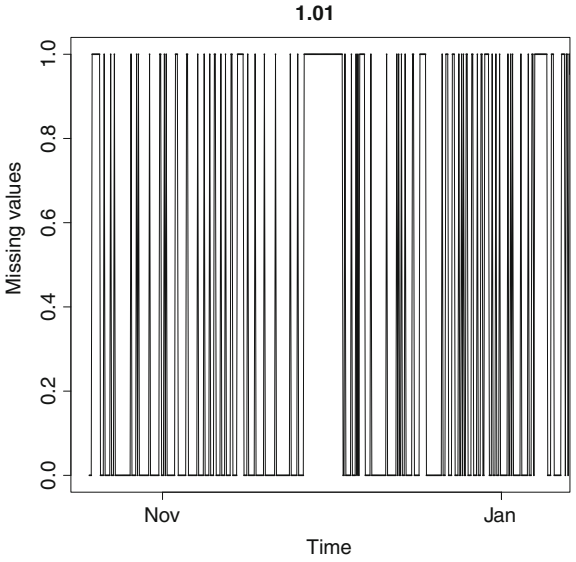

Fig. 10.6 The occurrence of missing locations over time for the first 500 locations: missing values are 1 and successful fixes are 0

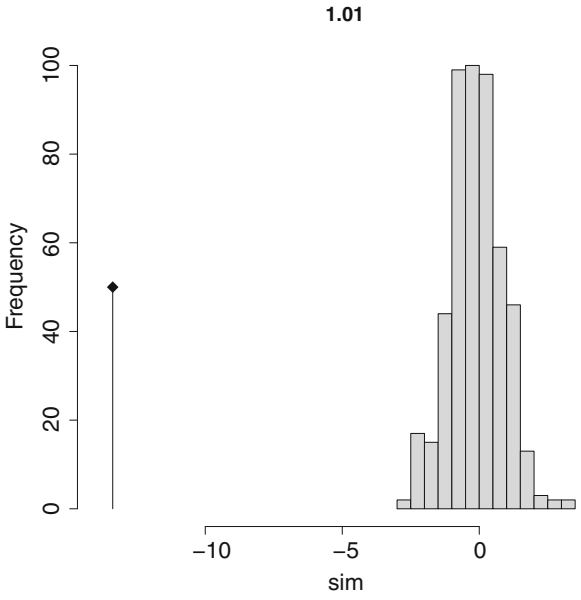

Fig. 10.7 Testing the temporal independence among missing locations. The *histogram* represents the expected distribution of missing values if they occurred independently. The pin on the *left* shows the observed distribution, which shows that missing values did not occur independently from each other

Figure 10.6 reveals that the missing values in November and December are not likely to occur independent of each other (you can verify this yourself for other periods by changing the limits of the x-axis with the xlim argument). You can test

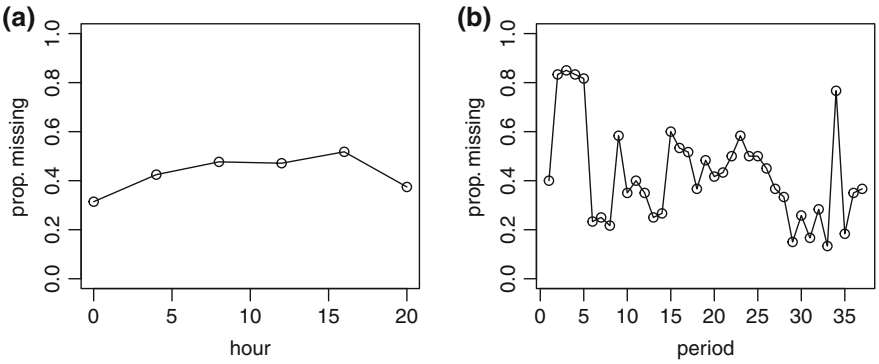

Fig. 10.8 Plot with the proportion missing locations for each hour of the day (panel **a**) and for each period of 10 days (panel **b**)

with the *runsNAltraj* function whether there is statistical significant clustering of the missing values:

```
runsNAltraj(ltrj[1])  #indeed, as the figures showed, it is not random
```

Indeed, Fig. 10.7 shows that for this trajectory, there is significant clustering of the missing fixes (you can test yourself whether this is also the case for the other animals). Thus, when you have one missing location, it is more likely that the next location will be missing too. Such temporal dependence in the probability to obtain fixes is not surprising, because the conditions affecting this probability are likely temporally autocorrelated. For instance, it is known that for a GPS receiver, it is more difficult to contact the satellites within dense forests, and so when an animal is in such a forest at time t, it is more likely to be still in this forest at time $t + 1$ than at time $t + 2$, thus causing temporal dependence in the fix-acquisition probability. Unfortunately, as said, this temporal dependence in the 'missingness' of locations holds the risk of introducing biases in the results of your analysis.

Visual inspection of figures like Fig. 10.6 can help the assessment of whether the temporal dependence in missing locations will have large effects on the analysis. Other figures can also help this assessment. For instance, you can plot the number of missing locations for each hour of the day, or for periods of the year (e.g. each week or month):

```
par(mfrow=c(1,2))
plot(c(0, 4, 8, 12, 16, 20),
     tapply(ltrj[[1]]$x, as.POSIXlt(ltrj[[1]]$date)$hour,
     function(x)mean(is.na(x))),
     xlab="hour", ylab="prop. missing", type="o", ylim=c(0,1), main="a")
periods <- trunc(as.POSIXlt(ltrj[[1]]$date)$yday/10)
plot(tapply(ltrj[[1]]$x, periods, function(x)mean(is.na(x))),
     xlab="period", ylab="prop. missing", type="o", ylim=c(0,1), main="b")
```

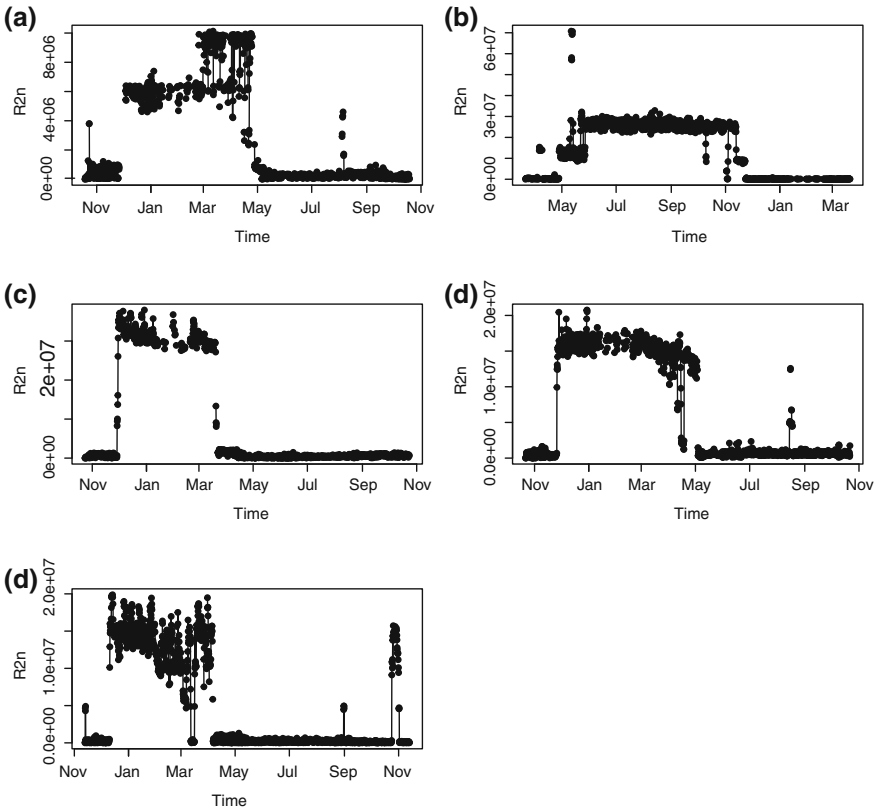

Fig. 10.9 The NSD for each individual

Figure 10.8 shows that there is no strong bias in the time of the day; you can therefore be fairly confident that additional results will not be biased regarding the diurnal cycle. However, there are four consecutive blocks of 10 days with low fix-acquisition rate, which could be an issue. Fortunately, the other periods during the winter are providing us with enough data. You therefore expect bias on your results to be minimal. In cases when there are longer periods with missing data, it can be necessary to balance the data. It is obviously not straightforward to create new data; however, you can remove random locations in periods when you have 'too many' observations. In our demonstration, we will proceed without further balancing the data.

The NSD is a commonly used metric for animal space use; it is the squared straight-line distance between each point and the first point of the trajectory (see Fig. 10.2b). It is a very useful metric to assess, for instance, the occurrence of seasonal migration patterns. An animal that migrates between summer and winter ranges will often show a characteristic hump shape in the NSD, as exemplified clearly in Fig. 10.9:

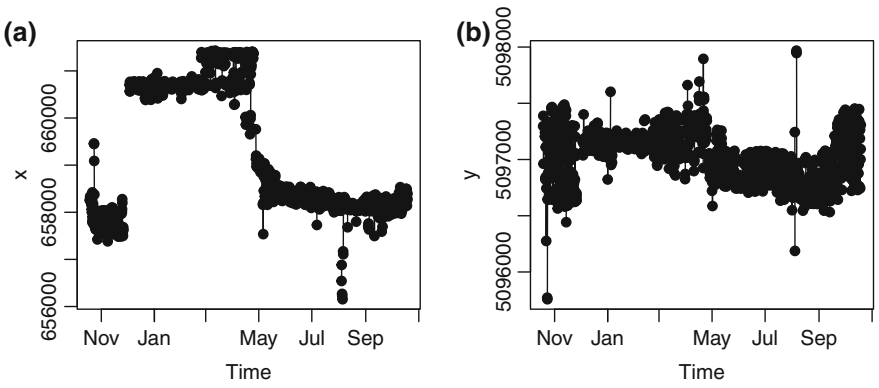

Fig. 10.10 The x- and y-axis against time in panel **a** and **b**, respectively, for the first individual

```
plotltr(ltrj, which = "R2n")
```

The NSD profiles in Fig. 10.9 strongly suggest the occurrence of seasonal migration in all five animals. During the summer (from May till November), the animals seem to be in one area and during the winter (from December till April) in another. The NSD is a one-dimensional representation of the animal's space use, which facilitates the inspection of it against the second dimension of time. On the other hand, it removes information present in the two-dimensional locations provided by the GPS. Relying exclusively on the NSD can in certain situations give rise to wrong inferences; we therefore highly recommend also inspecting the locations in two dimensions. One of the disadvantages of the NSD is that the starting point is often somewhat arbitrary. It can help to use a biological criterion such as the fawning period to start the year.

As an alternative for (or in addition to) the NSD, you can plot both spatial dimensions against time as in Fig. 10.10:

```
par(mfrow=c(1,2))
plot(ltrj[[1]]$date, ltrj[[1]]$x, pch=19, type="o", xlab="Time",
     ylab="x", main="a")
plot(ltrj[[1]]$date, ltrj[[1]]$y, pch=19, type="o", xlab="Time",
     ylab="y", main="b")
```

To avoid the intrinsic reduction of information by collapsing two dimensions into one single dimension, you also plot both spatial dimensions and use colour to depict the temporal dimension:

```
ltrj2 <- na.omit.ltraj(ltrj)
par(mfrow=c(3,2))
for (i in c(1:5))
{plot(ltrj2[[i]][c("x","y")], col=rainbow(12)[as.POSIXlt(ltrj2[[i]]$date)$mon+1],
      pch=19, asp=T)
 segments(ltrj2[[i]]$x[-nrow(ltrj[[i]])], ltrj2[[i]]$y[-nrow(ltrj2[[i]])],
          ltrj2[[i]]$x[-1],ltrj2[[i]]$y[-1],
          col=rainbow(12)[as.POSIXlt(ltrj2[[i]]$date[-1])$mon+1])
 title(c("a", "b", "c", "d", "e")[i])
 }
plot(c(0:100),c(0:100), type="n", xaxt="n", yaxt="n", xlab="", ylab="", bty="n")
mon <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
         "Oct", "Nov", "Dec")
legend(x=20, y=90, legend=mon[c(0:5)+1], pch=19, col=rainbow(12)[1:6], bty="n")
legend(x=60, y=90, legend=mon[c(6:11)+1], pch=19, col=rainbow(12)[7:12], bty="n")
```

Figure 10.11 confirms our interpretation of Fig. 10.9. All five animals have at least two seasonal centres of activity: winter versus summer. The movement between these centres occurs around November and around April. The comparison of Figs. 10.9 and 10.11 reveals easily the respective strengths of both figures. It is easier to read from Fig. 10.9 the timing of events, but it is easier to read from Fig. 10.11 the geographic position of these events. This demonstrates the importance of making several figures to explore the data.

Now that you have familiarised yourselves with the structure of the data and have ensured that your data are appropriate for the analysis, you can proceed with answering your ecological questions in the following sections.

Home Range Estimation

A home range is the area in which an animal lives. In addition to Burt's (1943) aforementioned definition of the home range as 'the area an animal utilizes in its normal activities', Cooper (1978) pointed out that a central characteristic of the home range is that it is temporally stable. Our previous exploration of the data has shown that the space use of our roe deer is not stable. Instead, it seems to consist of a migration between two seasonal home ranges. Figures 10.9 and 10.10 suggest that space use within these seasonal ranges is fairly stable. It is thus clear that the concept of a home range is inherently tied to a time frame over which space use was fairly stable, in our case two seasons.

An animal's home range has been quantified by the concept of the 'utilization distribution (UD)'. Van Winkle (1975) used the term UD to refer to 'the relative frequency distribution for the points of location of an animal over a period of time'. The most common estimator for the UD is the kernel density estimator. In Fig. 10.12, we remind the reader of the general principle underlying such analysis. Several methods for home range computation are implemented in the

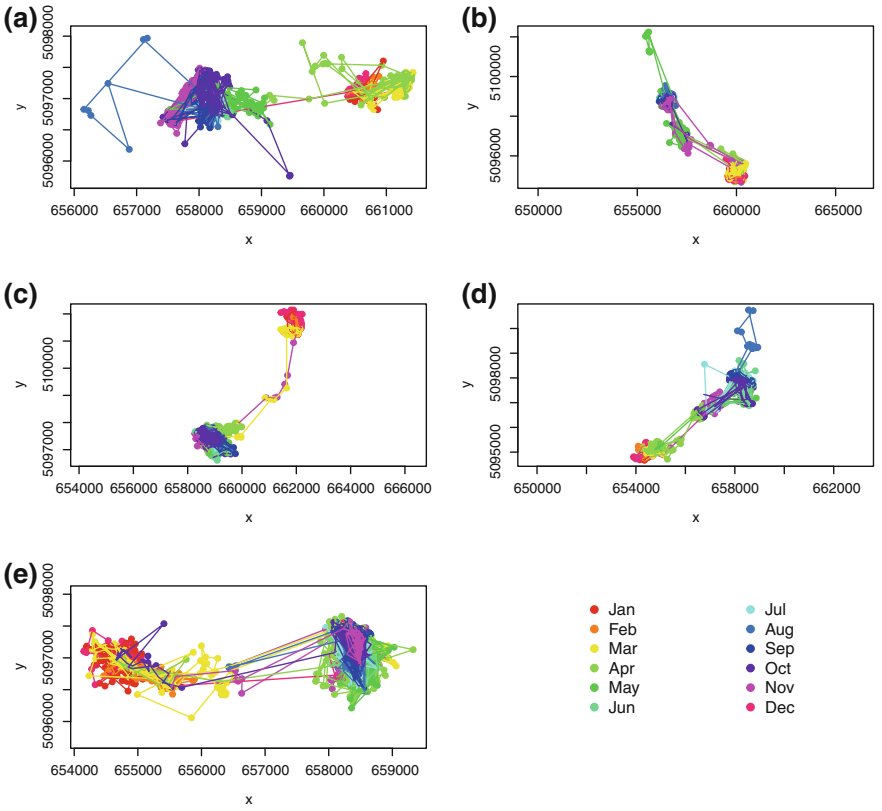

Fig. 10.11 Coloured trajectories for all individuals, locations from each month are coloured differently

'*adehabitatHR*' library; the *kernelUD* function calculates the kernel utilization density from 2D locations:

```
library(adehabitatHR)
library(sp)
trj <- ld(ltrj)
trj <- trj[!is.na(trj$x),]
(kUD <- kernelUD(SpatialPointsDataFrame(trj[c("x","y")],
                 data=data.frame(id=trj$id)),h=100,grid=200,kern="epa"))
image(kUD[[1]])
```

The function *kernelUD* requires a *SpatialPoints* object or a *SpatialPointsDataFrame*, and it returns a *SpatialPixel* object, adehabitat relies on the spatial classes from the '*sp*' library. A familiarity with the spatial classes from '*sp*' will

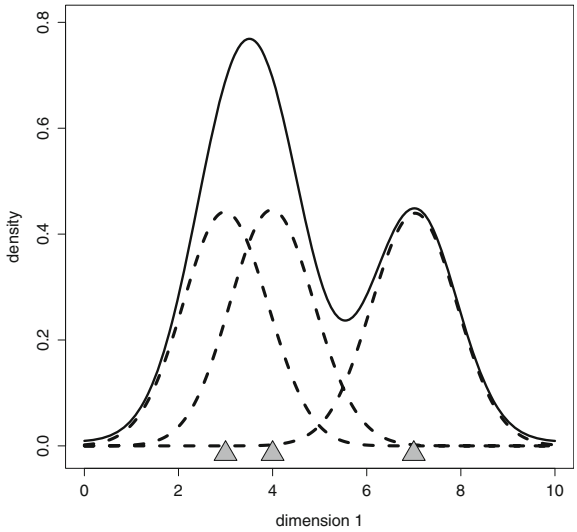

Fig. 10.12 A one-dimensional example of a kernel density estimator for three points. For each of the observations (i.e. 3, 4 and 7), a distribution (e.g. *Gaussian curve*) is placed over them (the *dashed lines*); these distributions are aggregated to obtain the cumulative curve (the *full black line*). This cumulative curve is the kernel density estimator of these points. The width of initial distributions used is the smoothing factor and is a parameter the researcher has to select. Kernel home range estimation works in a similar way in 2D with for instance a bivariate normal distribution to smooth the locations

therefore be helpful for your analysis of animal space use data in R (see Bivand et al. 2008, or the vignettes available for '*sp*'). You create the *SpatialPointsDataFrame* from a dataframe, which you obtained from the *ltrj* using the *ld* function (i.e. list to dataframe). Note: '*SpatialPoints*' cannot contain missing coordinates; therefore, you keep only those rows where you have no missing values for the x coordinate (i.e. *!is.na(trj$x)*, the '!' means 'not' in R). The resulting pixel map in Fig. 10.13 shows the areas that are most intensely used by this individual.

During our data exploration, we found that our roe deer occupy a separate summer and winter range: Are both ranges of similar size? For this, you have to compute the home range separately for summer and winter. In Fig. 10.9, you can see that the summer range is occupied at least from day 150 (beginning of June) to day 300 (end of October) and that the winter range is occupied from days 350 (mid-December) till 100 (end of March). You can use these dates to split compute the kernel for summer and winter separately; the function *kernel.area* computes the area within a percentage contour. We demonstrate the computation for the 50, 75 and 95 %:

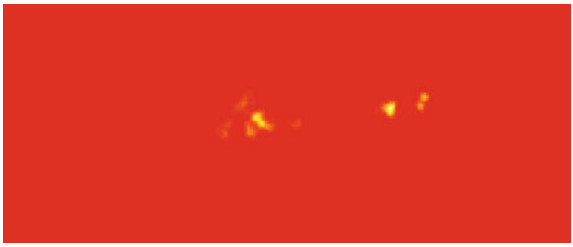

Fig. 10.13 Kernel UD of the first individual, in *yellow*, is the intensely used areas

```
trj$yday <- yday(trj$date)
trj$season <- ifelse(trj$yday>150 & trj$yday<300, "summer", NA)
trj$season <- ifelse(trj$yday>350 | trj$yday<100, "winter", trj$season)
trj <- trj[!is.na(trj$season),]
(kUD <- kernelUD(SpatialPointsDataFrame(trj[c("x","y")],
                 data=data.frame(id=paste(trj$id, trj$season))),
                 h =100, grid=200, kern = "epa"))

## ********** Utilization distribution of several Animals ************
##
## Type: probability density
## Smoothing parameter estimated with a  specified smoothing parameter
## This object is a list with one component per animal.
## Each component is an object of class estUD
## See estUD-class for more information

area <- kernel.area(kUD, percent = c(50, 75, 95), unin = "m", unout = "ha")
(areas <- data.frame(A50=unlist(area[1,]), A75=unlist(area[2,]),
                     A95=unlist(area[3,]), id=rep(c(1:5), each=2),
                     seas=rep(c("S", "W"), 5)))

##              A50   A75    A95 id seas
## X1.summer  9.931 22.51  52.30  1    S
## X1.winter  6.902 14.19  33.06  1    W
## X2.summer 10.915 21.92  40.47  2    S
## X2.winter 11.277 27.07  62.65  2    W
## X3.summer 12.977 30.24  62.96  3    S
## X3.winter  7.873 19.79  50.84  3    W
## X4.summer 10.230 24.25  83.17  4    S
## X4.winter  5.747 11.14  28.43  4    W
## X5.summer 12.498 24.31  49.14  5    S
## X5.winter 18.344 41.74 112.72  5    W
```

You can visualise these results clearly with boxplots:

```
par(mfrow = c(1, 3))
boxplot(areas$A50 ~ areas$seas, cex = 2, main = "a")
boxplot(areas$A75 ~ areas$seas, cex = 2, main = "b")
boxplot(areas$A95 ~ areas$seas, cex = 2, main = "c")
```

Figure 10.14 shows that more than a change in the mean range size, there seems to be a marked change in the individual variation between seasons. During the winter season, there seems to be much larger individual variation in range size than there is in summer; however, more data will be required to further investigate this seasonal variation in range sizes.

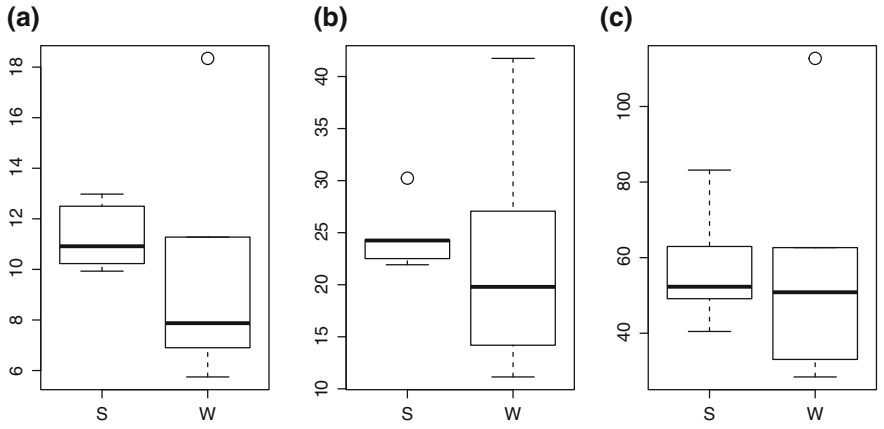

Fig. 10.14 The boxplots of the areas (in ha) of the seasonal ranges from left to right the 50, 75 and 95 % kernel contours. Each panel depicts summer (S) on the *left*, and winter (W) on the *right*

Habitat Use and Habitat Selection Analysis

In the previous exercise, you saw that roe deer change the location of their activities from summer to winter. Such seasonal migrations are often triggered by changes in the environment. You can then wonder which environmental conditions are changing when the animal moves between its seasonal ranges. For instance, snowfall is a driver for many migratory ungulates in northern and alpine environments, and winter ranges are often characterised by less snow cover than the summer ranges in winter (e.g. Ball et al. 2001). Thus, you would expect that roe deer will be moving down in altitude during the winter to escape from the snow that accumulates at higher altitudes.

If roe deer move to lower altitudes during winter, then they probably also move closer to roads, which are usually found in valley bottoms. You would not necessarily expect roe deer to show a seasonal response directly toward roads, but you do expect this as a side effect from the shift in altitude. Such closer proximity to roads can have substantial effects on road safety, as animals close to roads are at a higher risk of road crossings and thus traffic accidents. Ungulate vehicle collisions are an important concern for road safety and animal welfare. From an applied perspective, it is thus an interesting question to see whether there is a seasonal movement closer to roads, which could partly explain seasonal patterns often observed in ungulate vehicle collisions.

You first add the environmental data to your inspected trajectory with the *merge* function:

```
trj <- ld(ltrj)
trj <- merge(trj, locs, by.x=c("id", "date"),
             by.y=c("animals_id", "UTC_time"), all.x=T)
```

You inspect then whether there is a relationship between the distance to the roads and the altitude:

```
library(lattice)
xyplot(roads_dist ~ altitude_srtm| factor(id),
       xlab = "altitude", ylab = "dist. road", col = 1,
       strip = function(bg = "white", ...) strip.default(bg = "white", ...),
       data = trj)
```

Figure 10.15 shows an interesting relationship between altitude and distance to roads. Each individual shows two clusters, which are possibly corresponding with the two seasonal ranges you detected before. Within each cluster, there is a positive relationship between altitude and distance to roads; i.e. at higher altitudes, the distance to roads is greater. However, when you compare both clusters, it seems that the cluster at higher altitude is often also closer to roads (except for Animal 1). Overall, it seems that in your data, there is no obvious positive relationship between altitude and distance to roads.

Let us now investigate the hypothesis that there is a seasonal change in roe deer altitude:

```
xyplot(altitude_srtm ~ as.POSIXlt(acquisition_time)$yday | factor(id),
       xlab = "day of year", ylab = "altitude", col = 1,
       strip = function(bg = "white", ...) strip.default(bg = "white", ...),
       data = trj)
```

Figure 10.16 shows that there are marked seasonal changes in the altitude of the roe deer positions. As you expected, roe deer are at lower altitudes during the winter than they are during the summer. This pattern explains the occurrence of the two clusters of points for each individual in Fig. 10.15.

You can now proceed by testing the statistical significance of these results. You use the same seasonal cutoff points as before:

```
trj$yday <- yday(trj$date)
trj$season <- ifelse(trj$yday > 150 & trj$yday < 300, "summer", NA)
trj$season <- ifelse(trj$yday > 350 | trj$yday < 100, "winter", trj$season)
trj2 <- trj[!is.na(trj$season), ]
fit <- lm(altitude_srtm ~ as.factor(season), data = trj2)
```

The function *lm* is used to fit a linear model to the data (note: for this simple case a Student's *t* test would have been sufficient).

These results show that as expected the roe deer move to lower altitudes during the winter:

	Estimate	Std. error	*t* value	Pr(>|*t*|)
(Intercept)	1,581.6609	2.6212	603.41	0.0000
as.factor(season)winter	−572.3900	4.2519	−134.62	0.0000

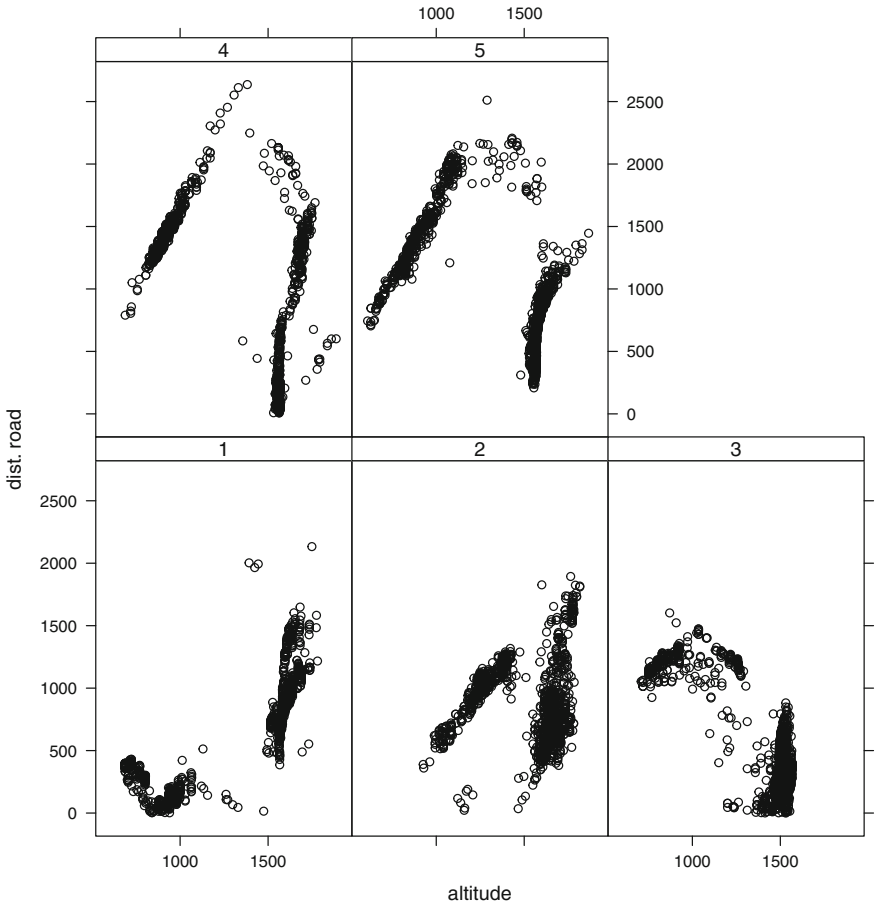

Fig. 10.15 The changing distance to roads as a function of the altitude for each individual

In winter, the roe deer are on average 572 m lower than during the summer, which is a 33 % decrease.

A treatment on model validation falls outside the scope of this book. We refer the reader to introductory books in statistics; several such books are available using examples in R (e.g. Crawley 2005; Zuur et al. 2007).

In this example, you have focused on the habitat use of roe deer. Often researchers are not only interested in the habitat characteristics used by the animals, but also in the comparison between use and availability—i.e. habitat selection. In habitat-selection studies, the used habitat characteristics are compared against the characteristics the animal could have used or the available habitat. Thus, to perform habitat-selection analysis, you have to sample from the available points and obtain the habitat characteristics for these points.

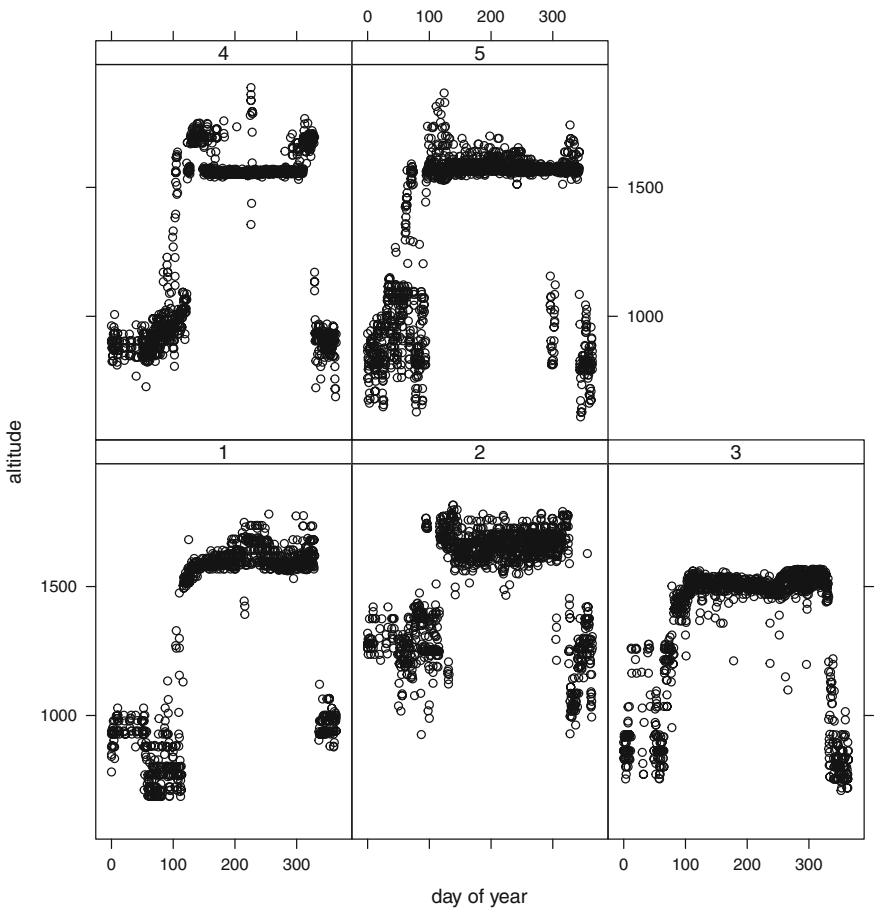

Fig. 10.16 The altitude of roe deer locations as a function of the day of the year. You see marked seasonal changes

The available locations are commonly sampled randomly at two scales: within the study area to study home range placement, or within the individual home range to study home range use (respectively, called second- and third-order habitat selection following Johnson 1980). We will demonstrate third-order habitat selection and use a minimum convex polygon (MCP) to characterise the area available for each roe deer, from which we sample 2,000 random locations. The *mcp*-function in R requires the use of a *SpatialPointsDataFrame* when using multiple animals (for a single animal a *SpatialPoints* object suffices):

```
trj2 <- na.omit(trj[,c("id", "x","y")])
sptrj <- SpatialPointsDataFrame(SpatialPoints(trj2[,c("x","y")]),
                                data=data.frame(id=trj2$id))
ranges <- mcp(sptrj, percent=100)
```

Then, you sample for each individual randomly from its available range, and you place the coordinates from these sampled locations together with the animal's id in a *data.frame*. Operations like this, in which something needs to be repeated for a number of individuals, are easily performed using the *list* format, which is also the reason that '*adehabitatLT*' uses a *list* to store trajectories. However, a database works with *data.frames*; therefore, you will have to bind the *data.frames* in the list together in one large *data.frame*.

```
set.seed(0) #to ensure replication of the same randomly sampled points
rndpts <- lapply(c(1:5),
              function(x, ranges){spsample(ranges[ranges@data$id==x,], n=2000,
                                  type="random", iter=100)},ranges=ranges)
rndpts <- lapply(c(1:5),}{function(x,rndpts)data.frame(rndpts[[x]]@coords,id=x),
                  rndpts=rndpts)
rndpts <- do.call("rbind", rndpts)
```

Figure 10.17 shows the areas you considered available for each individual roe deer, from which you sampled the random locations:

```
plot(ranges)
points(rndpts[c(1:100), c("x", "y")], pch = 16)  #shows the first 100 random points
```

The easiest way to obtain the environmental data for these random points is to simply upload them into the database and extract the required information from there. To facilitate the ordering of your random locations, you add a column *nb* numbered 1 to the number of random points. The function *dbWriteTable* writes a table to the database; you can specify a schema (*analysis*) in addition to the table name (*rndpts_tmp*):

```
rndpts$nb <- c(1:nrow(rndpts))
dbWriteTable(con, c("analysis", "rndpts_tmp"), rndpts)

## [1] TRUE
```

Next, you use the database to couple the locations in this table to the environmental data stored in the database. The easiest way to do this is by first adding a geometry column for the random locations:

```
dbSendQuery(con,
  "ALTER TABLE analysis.rndpts_tmp ADD COLUMN geom geometry(point, 4326);")

## <PostgreSQLResult:(6096912,1,7)>

dbSendQuery(con,
  "UPDATE analysis.rndpts_tmp
   SET geom = st_transform((st_setsrid(st_makepoint(x,y),23032)),4326);" )

## <PostgreSQLResult:(6096912,1,8)>
```

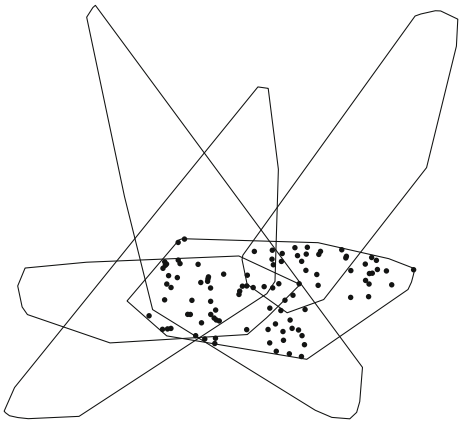

Fig. 10.17 The available areas for each roe deer estimated by a MCP. The first 100 locations sampled randomly from the area of individual 1 are represented by *black dots*

You extract the altitude and land cover for the random locations from the rasters stored in the database with the following queries (the details of these SQL queries were discussed earlier in this book):

```
altitude <- fetch(dbSendQuery(con,
    "SELECT st_value(srtm_dem.rast, geom) as altitude
     FROM env_data.srtm_dem, analysis.rndpts_tmp
     WHERE st_intersects(srtm_dem.rast,geom) ORDER BY nb;"), -1)
landcover <- fetch(dbSendQuery(con,
    "SELECT st_value(corine_land_cover.rast, st_transform(geom,3035)) as landcover
     FROM env_data.corine_land_cover, analysis.rndpts_tmp
     WHERE st_intersects(corine_land_cover.rast, st_transform(geom,3035))
                                                          ORDER BY nb;"), -1)
```

You extract the distance to the closest road for the random locations from the roads stored in the database with the following query (this query can require a few minutes):

```
mindist <- fetch(dbSendQuery(con,
    "SELECT min(distance) as dist_roads
     FROM (SELECT nb, st_distance(roads.geom::geography,
           rndpts_tmp.geom::geography)::integer as distance
           FROM env_data.roads, analysis.rndpts_tmp) AS a
           GROUP BY a.nb ORDER BY nb;"), -1)
```

You add these environmental data to the randomly sampled available locations:

```
rndpts <- cbind(rndpts, altitude, landcover, mindist)
head(rndpts)

##        x       y id nb altitude landcover dist_roads
## 1 660909 5097377  1  1      940        23         30
## 2 657779 5096660  1  2     1650        25       1555
## 3 658739 5096304  1  3     1778        24       1500
## 4 657638 5097232  1  4     1678        25       1135
## 5 658984 5097551  1  5     1510        18        289
## 6 659459 5097810  1  6     1459        18         82
```

Now that you no longer need these locations in the database, you can remove the table:

```
dbRemoveTable(con, c("analysis", "rndpts_tmp"))
```

```
## [1] TRUE
```

A discussion on habitat selection falls outside the scope of this chapter. More information on exploratory habitat selection using R can be found in the *vignette(adehabitatHS)*; for a general discussion on the use of resource selection functions on habitat selection, we refer the reader to the book by Manly et al. (2002). We will merely visualise the difference in the habitat types between the used and the available locations.

To ensure proper comparison of used and available habitats, you provide the same levels to both data sets. Moreover, to allow our seasonal comparison, you also need to allocate the random locations to the summer and winter season:

```
trj <- na.omit(trj[,c("id", "x", "y", "roads_dist","corine_land_cover_code",
                      "altitude_srtm", "season")])
names(trj) <- c("id", "x", "y", "dist_roads", "landcover", "altitude", "season")
trj$landcover <- factor(trj$landcover, levels=c(18, 21, 23:27, 29, 31, 32))
#allocate the random locations to each season
rndpts$season <- c("summer", "winter")
rndpts$landcover <- factor(rndpts$landcover, levels=c(18, 21, 23:27, 29, 31, 32))
```

Now, you will compute for each individual the number of locations inside each habitat type:

```
use_win <- table(trj$id[trj$season=="winter"],
                 trj$landcover[trj$season=="winter"])
ava_win <- table(rndpts$id[rndpts$season=="winter"],
                 rndpts$landcover[rndpts$season=="winter"])
use_sum <- table(trj$id[trj$season=="summer"],
                 trj$landcover[trj$season=="summer"])
ava_sum <- table(rndpts$id[rndpts$season=="summer"],
                 rndpts$landcover[rndpts$season=="summer"])
#determine the proportions of each class
calc.prop <- function(x){(x/sum(x))}
use_win <- t(apply(use_win, 1, calc.prop))
ava_win <- t(apply(ava_win, 1, calc.prop))
use_sum <- t(apply(use_sum, 1, calc.prop))
ava_sum <- t(apply(ava_sum, 1, calc.prop))
#to avoid division by zero, we add a small number to each element in the table
use_win <- use_win+0.01
ava_win <- ava_win+0.01
use_sum <- use_sum+0.01
ava_sum <- ava_sum+0.01
library(adehabitatHS)
sr_win <- widesIII(use_win,ava_win, avknown = FALSE, alpha = 0.05)
sr_sum <- widesIII(use_sum,ava_sum, avknown = FALSE, alpha = 0.05)
```

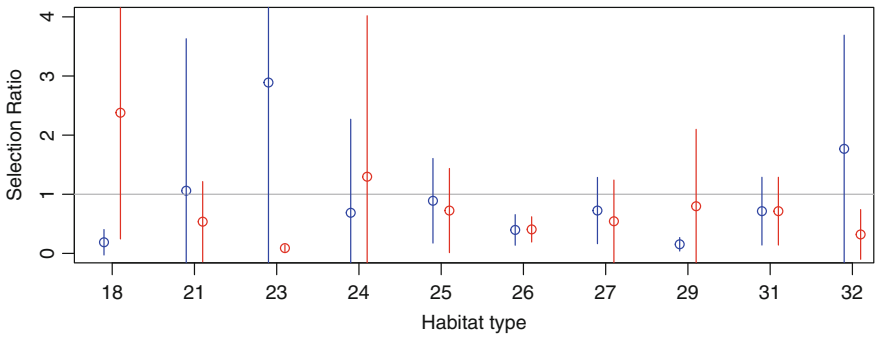

Fig. 10.18 The ratio of the proportion used and the proportion available (known as the selection ratio). Values above 1 are used more than their availability and vice versa. Blue points are for winter and red for summer. You can find the corresponding habitat types in the corine_land_cover_legend table from your database

The function *widesIII* in the '*adehabitatHS*' library computes the selection ratios for individually tracked animals; it also provides a number of statistical tests.

```
plot(c(1:10)-0.1, sr_win$wi, xaxt="n", ylab="Selection Ratio",
     xlab="Habitat type", col="blue", ylim=c(0,4))
axis(1, at=c(1:10), labels=c(18, 21, 23:27, 29, 31, 32))
abline(h=1, col="dark grey")
points(c(1:10)+0.1, sr_sum$wi, col="red")
segments(c(1:10)-0.1, sr_win$ICwiupper, c(1:10)-0.1, sr_win$ICwilower,col="blue")
segments(c(1:10)+0.1, sr_sum$ICwiupper, c(1:10)+0.1, sr_sum$ICwilower,col="red")
```

For a more extensive discussion of selection ratios, we refer you to the aforementioned references. Here, you limit yourselves to visualising the selection ratios for both seasons. Figure 10.18 shows that the roe deer seems to select more for pastures (class 18) during summer, whereas for most roe deer, their use of broad-leaved forests seem to be higher during the winter months (class 23). Great care should be taken not to over-interpret the unreliable results from classes that are hardly present in the study area (e.g. classes 26–31). These types of figures and tables are highly suitable to inspecting categorical data such as land cover. Continuous data such as altitude are better represented using histograms.

In the previous demonstrations, you have been using R to visualise and analyse data stored in the database, and you also used R as an interface to interact with the database. An alternative approach is to use R from within the database, which we will demonstrate in the next chapter.

Acknowledgments M. Basille provided valuable comments on the earlier drafts of this chapter. The work for this chapter was funded by Norwegian Research Council project number 208434-F40 'Management of biodiversity and ecosystem services in spatially structured landscapes'.

References

Ball JP, Nordengren C, Wallin K (2001) Partial migration by large ungulates: characteristics of seasonal moose Alces alces ranges in northern Sweden. Wild Biol 1:39–47

Bivand RS, Pebesma EJ, Rubio VG (2008) Applied spatial data: analysis with R. Springer, New York

Burt WH (1943) Territoriality and home range concepts as applied to mammals. J Mammal 24(3):346–352

Calenge C (2006) The package 'adehabitat' for the R software: a tool for the analysis of space and habitat use by animals. Ecol Model 197(3):516–519

Calenge C, Dray S, Royer-Carenzi M (2009) The concept of animals' trajectories from a data analysis perspective. Ecol Inform 4(1):34–41

Cooper WE Jr (1978) Home range criteria based on temporal stability of areal occupation. J Theor Biol 73(4):687–695

Crawley MJ (2005) Statistics: an introduction using R. Wiley, New York

Fortin D, Beyer HL, Boyce MS, Smith DW, Duchesne T, Mao JS (2005) Wolves influence elk movements: behavior shapes a trophic cascade in Yellowstone National Park. Ecology 86(5):1320–1330

Johnson DH (1980) The comparison of usage and availability measurements for evaluating resource preference. Ecology 61(1):65–71

Kie JG, Matthiopoulos J, Fieberg J, Powell RA, Cagnacci F, Mitchell MS, Moorcroft PR (2010) The home-range concept: are traditional estimators still relevant with modern telemetry technology? Philos Trans R Soc B: Biol Sci 365(1550):2221–2231

Manly BFJ, McDonald LL, Thomas DL, McDonald TL, Erickson WP (2002) Resource selection by animals: statistical analysis and design for field studies. Kluwer, The Netherlands

Turchin P (1998) Quantitative analysis of movement: measuring and modeling population redistribution in animals and plants. Sinauer Associates, Sunderland

Van Winkle W (1975) Comparison of several probabilistic home-range models. J Wild Manag 39:118–123

Zuur AF, Ieno EN, Smith GM (2007) Analysing ecological data. Springer, New York

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
```

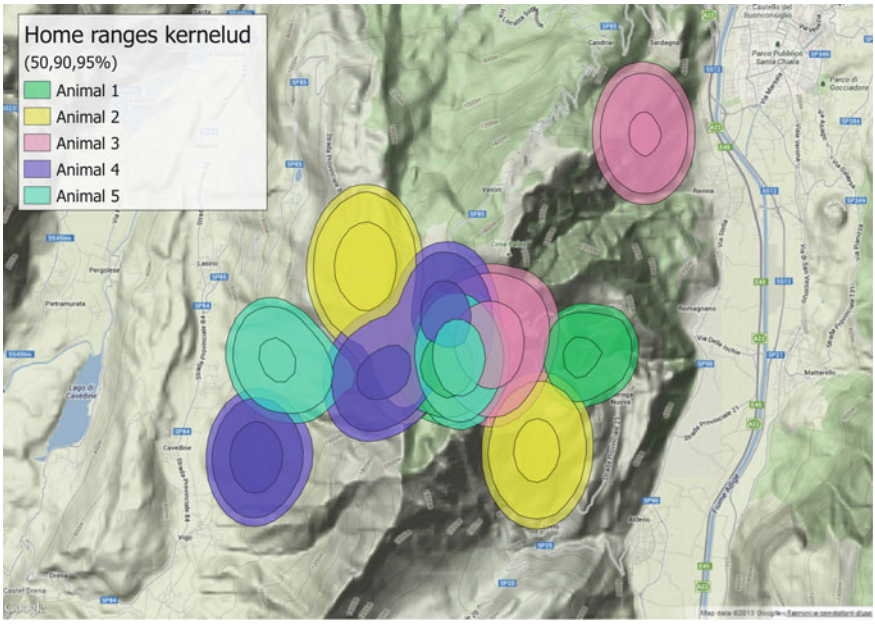

Fig. 11.1 Kernel home ranges at 50, 90 and 95 %

```
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
```

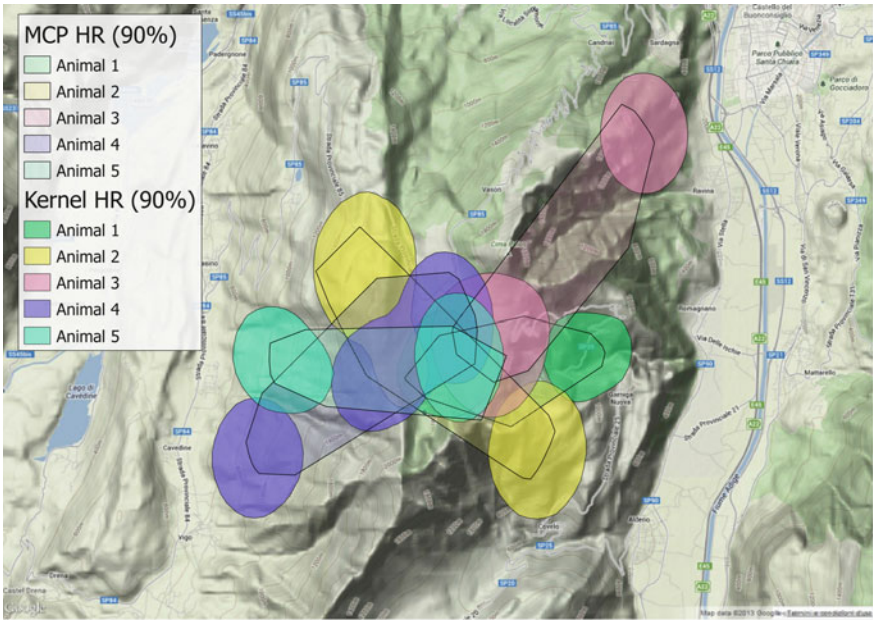

Fig. 11.2 Comparison between kernel and MCP home range at 90 %

```
    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/.

Chapter 12
Deciphering Animals' Behaviour: Joining GPS and Activity Data

Anne Berger, Holger Dettki and Ferdinando Urbano

Abstract In the previous chapters, you have exclusively worked with GPS position data. We showed how to organise these data in databases, how to link them to environmental data and how to connect them to R for further analysis. In this chapter, we introduce an example of data recorded by another type of sensor: acceleration data, which can be measured by many tags where they are associated with the GPS sensors and are widely used to interpret the behaviour of tagged animals. The general structure of these data and an overview of possibilities for analysis are given. In the exercise for this chapter, you will learn how to integrate an acceleration data set into the database created in the previous chapters and link it with other information from the database. At the end, the database is extended with acceleration data and with an automated procedure to intersect these layers with GPS positions.

Keywords Accelerometer · Activity analysis · Data management · Spatial database

A. Berger (✉)
Leibniz Institute for Zoo and Wildlife Research, Alfred-Kowalke-Straße 17,
D-10315 Berlin, Germany
e-mail: berger@izw-berlin.de

H. Dettki
Umeå Center for Wireless Remote Animal Monitoring (UC-WRAM),
Department of Wildlife, Fish, and Environmental Studies,
Swedish University of Agricultural Sciences (SLU), Skogsmarksgränd,
SE-901 83 Umeå, Sweden
e-mail: holger.dettki@slu.se

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

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_12,

Introduction

In the previous chapters, you learned how to correlate GPS positions with other spatiotemporal information such as NDVI values and DEMs. However, many kinds of bio-logging sensors are available to record a large set of information related to animals. In fact, we are quickly moving from animals monitored by one single sensor, usually a GPS receiver, to animals monitored by multiple, integrated sensors that register spatial (i.e. GPS positions) and non-spatial measurements such as acceleration, temperature or GSM signal quality. In recent years, commercial solutions have emerged to deploy a camera on animals, or even internal sensors in the animals' body to register heartbeat and body temperature. These sensors are usually integrated on a unique physical support (the collar or tag). Data from all these different sensors can be related to the spatial position of the animal and to each other on the basis of the acquisition time of the recorded information, thus giving a complete picture of the animal at a certain time. This integrated set of information can be used to fully decipher the animals' behaviour in space and time. The opportunity to answer new biological questions through the information derived from these multi-sensor monitoring platforms implies a number of further challenges in terms of data management. To be able to explore the multifaceted aspects of animals' behaviour, researchers must deal with even bigger and more diverse sets of data that require a more complex database data model and data acquisition procedures.

A complete discussion of methods to integrate data from all the available bio-logging sensors is outside the scope of this book. We will use acceleration data as an example to illustrate how you can include other sensor data into the database. One of the most-used non-spatial sensors in animal ecology is the acceleration sensor (accelerometer), often called an 'activity' sensor, which measures the acceleration of the body where the sensor is fixed. In wildlife telemetry studies, the activity data measured by these accelerometers, in combination with the spatial position of the animal, are widely used for a range of purposes:

- To detect time of death (being the original function of activity sensors in wildlife GPS devices).
- In life-strategy investigations, to find general species-specific annual patterns and seasonal levels of activity parameters such as mean activity, day–night patterns, and the number and duration of activity or resting phases (e.g. Krop-Benesch et al. 2013).
- To distinguish behaviours by identification of behaviour-specific animal movement patterns using triaxial, or 3D, accelerometry (Shepard et al. 2008). Past studies have shown that triaxial accelerometry data collected from sensors on animals can be used to detect very subtle differences in the patterns of movement such as step counts, the distinction between different gaits (e.g. jump, gallop, trot and pace) or lameness and handicaps (Scheibe and Gromann 2006).
- To automatically detect specific behaviours such as calving, resting, fast locomotion or hunting (Löttker et al. 2009; Fröhlich et al. 2012). In combination

with the knowledge of the GPS position of the animal, it then will be possible to create a so-called functional habitat use map that says not only which places are used by the animal, but also what the animal is doing at those places or habitats. Since activity data typically have a higher temporal resolution than GPS positions, they also give more detailed information about the behaviour during the time between two GPS positions (e.g. resting behaviour, fast or slow locomotion) and a significantly finer movement pattern of the animal can be calculated than by using only GPS positions.

- To calculate energy expenditure and metabolic projections using 3D accelerometry and calibration for different conditions (Qasem et al. 2012).
- To non-invasively detect stress conditions on free-running wild animals through chronobiological time series analysis (Berger et al. 2003) or to quantify anthropogenic disturbances by controlled disturbance trials (Reimoser 2012).

Acceleration sensors measure acceleration by using mass inertia, sensing how much a mass presses on something when a force acts on it. In general, piezoelectric accelerometers are used in wildlife telemetry in which a crystal is attached to a mass, so when the accelerometer moves, the mass squeezes the crystal and generates a tiny electric voltage. Although this measurement procedure is essentially identical in all accelerometers that are used on animals, the acceleration sensors of various devices differ widely due to internal data processing methods, resolution and sensitivity. In addition, sensors can measure in one, two or three axes, with triaxial accelerometers providing three output signals—x, y and z—each for one of the three perpendicular axes.

Hence, the data obtained from accelerometers of different devices differ crucially from each other (e.g. 1D or 3D, different time intervals for measurements, recording absolute or mean values per time unit) and researchers must think carefully about what kind of measuring system, what measuring interval and what analysis should be used to properly address the study question. For instance, some specific analysis tasks (e.g. distinguishing between different behaviours) require high-resolution triaxial acceleration measurements that not all devices are capable of. The most important selection criteria between the different 'activity' sensors from different brands are the number of the measuring axes, the options for recording and storage interval settings, the type of internal data processing and the capabilities of data storage or data transmission. This means that for the practical management of a database, activity data vary greatly in their structure, depending on the used device or its sensor settings: they may have been measured at one, two or three axes, or by an all-round sensor, resulting in one, two or three values at measurement. Furthermore, there are activity data recorded and stored at the millisecond scale without any data processing (e.g. e-obs GPS devices[1]) and there are activity measurements for which the original data are processed within the tag

[1] e-Obs GmbH, http://www.e-obs.de/.

and the data output is in minutes (e.g. Vectronic Aerospace[2] or Lotek[3] devices). Regardless, no matter what activity sensors were used, you will always get the following output: for each measurement time (which could be milliseconds, seconds or minutes), there are one to three activity values (depending on the number of axes) that vary within their measurement range.

Import the Activity Data into the Database

The general structure of activity sensors and activity sensor data is similar to that of data from GPS sensors (see Chaps. 2, 3 and 4). The sensor is associated with (deployed on) an animal for a defined time range. In most of the cases, GPS and activity sensors are attached to the same support (e.g. a collar). In your data model, you have to clearly identify collars and sensors as separate (but related) objects. Different sensors attached to the same collar might have different activation periods, e.g. one sensor can stop working, while the other(s) continues to record information. In our case, we do not include the objects 'collars' in the database, in order to simplify the data model and because they have no additional interesting information associated with them. Other approaches are possible—see, for example, the data model proposed by Kranstauber et al. (2011).

The size of activity data sets can be orders of magnitude greater than GPS data sets. This might imply performance issues in terms of processing time and storage, which can suggest a different data management approach. For example, you might decide to keep a single table for activity data, joining together raw data and derived information (e.g. the identifier of the animal). You can also use raw data just in the import stage and then delete them from the database, using the plain text file downloaded from the sensors as backup. The best choice depends on the size of the database, the desired performance, the specific goals and the operational environment. In any case, as activity data are generally acquired periodically by radio (monthly or weekly) or just once per sensor through a cable when the sensor is physically recovered after being removed from (or falling off) the animal, you do not necessarily need to set up automatic, real-time procedures for data import (e.g. association with animal, creation of an acquisition timestamp from the time and date received from the collar, quality check). Finally, data often come in different formats from different activity sensors; thus, a specific data process might be necessary for each type of activity sensor.

For operational databases, keep in mind that when the size of a single table is very big, for example bigger than the RAM of your computer, it might be convenient to use partitioned table functionalities that split (behind the scenes) what is

[2] Vectronic Aerospace GmbH, http://www.vectronic-aerospace.com/wildlife.php.

[3] Lotek Inc., http://www.lotek.com/.

logically one large table into smaller physical pieces[4]. Another issue related to large tables is the use of external keys and indexes. These can result in a significant slowdown of the time necessary to import data. Again, you have to evaluate the best solution according to the size of your database, the frequency of updates and routine analysis, and the goals of your information system. In this exercise, we will keep it simple and use the same approach as for GPS data. Now, you have to extend the database structure to accommodate this new information. The tables needed are

- a table for information about activity sensors;
- a table for information about the deployment of the activity sensors on animals;
- a table for the raw data coming from activity sensors;
- a table for activity data associated with animals.

In the test data set (\tracking_db\data\sensors_data), you have a .csv file (ACTIVITY01508.csv) with the data collected by a Vectronic Aerospace GmbH activity sensor (ACTIVITY01508). It contains more than 100,000 records. You can explore the file content with a text editor. As you can see, you have eight attributes: the code of the activity sensor (created in the data acquisition step), the UTC and LMT time and date, the activity along the x- and y-axes, and the temperature. The x-axis measures acceleration in forward/backward motions as well as pitch angle by gravitational acceleration; the y-axis measures sideward as well as rotary motion using gravitational acceleration. Sensors are queried four times per second simultaneously on both axes. All measurements of each of the x- and y-axes are averaged over the user-selected sampling interval (here over 5 min) and given a value within a relative range between 0 and 255, characterising the mean x- and y-activities of each 5-min interval. Both sensors were physically associated with the same collar of the GPS sensor GPS01508. When you define the link between the activity sensor and the animal (deployment time range), this relationship will be explicitly defined by the fact that both GPS and activity sensors are activated on the same animal at the same time.

First of all, you create a new table to accommodate information about activity sensors:

```
CREATE TABLE main.activity_sensors(
  activity_sensors_id integer,
  vendor character varying,
  activity_sensors_code character varying,
  model character varying,
  insert_timestamp timestamp with time zone DEFAULT now(),
```

[4] http://www.postgresql.org/docs/9.2/static/ddl-partitioning.html.

```
  update_timestamp timestamp with time zone DEFAULT now(),
  CONSTRAINT activity_sensors_pkey
    PRIMARY KEY (activity_sensors_id),
  CONSTRAINT activity_sensors_code_unique
    UNIQUE (activity_sensors_code)
);
COMMENT ON TABLE main.activity_sensors
IS 'Catalogue of activity sensors.';

CREATE TRIGGER update_timestamp
  BEFORE UPDATE
  ON main.activity_sensors
  FOR EACH ROW
  EXECUTE PROCEDURE tools.timestamp_last_update();
```

Now, you can populate it. As you have a single activity sensor, you can directly insert the record:

```
INSERT INTO main.activity_sensors (activity_sensors_id, vendor,
activity_sensors_code, model)
  VALUES (1, 'Vectronic', 'ACTIVITY01508', 'Basic model');
```

You must now define a table to store the deployment time range of the activity sensor on an animal:

```
CREATE TABLE main.activity_sensors_animals(
  activity_sensors_animals_id serial,
  animals_id integer NOT NULL,
  activity_sensors_id integer NOT NULL,
  start_time timestamp with time zone NOT NULL,
  end_time timestamp with time zone,
  notes character varying,
  insert_timestamp timestamp with time zone DEFAULT now(),
  update_timestamp timestamp with time zone DEFAULT now(),
  CONSTRAINT activity_sensors_animals_pkey
    PRIMARY KEY (activity_sensors_animals_id ),
  CONSTRAINT activity_sensors_animals_animals_id_fkey
    FOREIGN KEY (animals_id)
    REFERENCES main.animals (animals_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT activity_sensors_animals_activity_sensors_id_fkey
    FOREIGN KEY (activity_sensors_id)
    REFERENCES main.activity_sensors (activity_sensors_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT time_interval_check
    CHECK (end_time > start_time)
);
COMMENT ON TABLE main.activity_sensors_animals
IS 'Table that stores information of deployments of activity sensors on
animals.';

CREATE TRIGGER update_timestamp
  BEFORE UPDATE
  ON main.activity_sensors_animals
  FOR EACH ROW
  EXECUTE PROCEDURE tools.timestamp_last_update();
```

As said previously, the association between activity data and animals will be performed by the operators; thus, no automatic procedure for the synchronisation of the content of this table to the activity data table is needed.

You can now populate the table (with a single record):

```
INSERT INTO main.activity_sensors_animals(animals_id, activity_sensors_id,
  start_time, end_time, notes)
  VALUES (3,1,'2005-10-23 20:00:53 +0','2006-10-28 13:00:00 +0','Death of
  animal. Sensor recovered.');
```

Next, you create the table to host the raw data coming from the activity sensor, replicating the structure of the .csv file. You add a field to store the complete acquisition time, joining time and date coming from the sensor. This process relies on the function *tools.acquisition_time_update*() already created for GPS data:

```
CREATE TABLE main.activity_data(
  activity_data_id serial NOT NULL,
  activity_sensors_code character varying,
  utc_date date,
  utc_time time without time zone,
  lmt_date date,
  lmt_time time without time zone,
  activity_x integer,
  activity_y integer,
  temp double precision,
  insert_timestamp timestamp with time zone DEFAULT now(),
  acquisition_time timestamp with time zone,
  CONSTRAINT activity_data_pkey
    PRIMARY KEY (activity_data_id ),
  CONSTRAINT activity_data_sensors_fkey
    FOREIGN KEY (activity_sensors_code)
    REFERENCES main.activity_sensors (activity_sensors_code)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION
);
COMMENT ON TABLE main.activity_data
IS 'Table that stores raw data as they come from the activity sensors (plus
the ID of the sensor).';

CREATE INDEX activity_acquisition_time_index
  ON main.activity_data
  USING btree (acquisition_time );
CREATE INDEX activity_sensors_code_index
  ON main.activity_data
  USING btree (activity_sensors_code);
CREATE TRIGGER update_acquisition_time
  BEFORE INSERT
  ON main.activity_data
  FOR EACH ROW
  EXECUTE PROCEDURE tools.acquisition_time_update();
```

Finally, you have to create a table to store activity data associated with animals:

```
CREATE TABLE main.activity_data_animals(
  activity_data_animals_id serial,
  activity_sensors_id integer,
  animals_id integer,
  acquisition_time timestamp with time zone,
  activity_x integer,
  activity_y integer,
  temp double precision,
  insert_timestamp timestamp with time zone DEFAULT now(),
  CONSTRAINT activity_data_animals_pkey
    PRIMARY KEY (activity_data_animals_id),
  CONSTRAINT activity_data_animals_animals_fkey
    FOREIGN KEY (animals_id)
    REFERENCES main.animals (animals_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT activity_data_animals_sensors_fkey
    FOREIGN KEY (activity_sensors_id)
    REFERENCES main.activity_sensors (activity_sensors_id)
    MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION
);
COMMENT ON TABLE main.activity_data_animals
IS 'Table that stores activity data associated with animals.';

CREATE INDEX activity_animals_acquisition_time_index
  ON main.activity_data_animals
  USING btree (acquisition_time );
CREATE INDEX activity_animals_id_index
  ON main.activity_data_animals
  USING btree (animals_id);
```

You can see a schematic representation of the relations between these new tables in Fig. 12.1.

You are now ready to import the raw data:

```
COPY main.activity_data(
  activity_sensors_code, utc_date, utc_time, lmt_date, lmt_time, activity_x,
  activity_y, temp)
FROM
  'C:\tracking_db\data\sensors_data\ACTIVITY01508.csv'
  WITH CSV HEADER DELIMITER ';';
```

The last step is the association of activity data with the animal in the table *main.activity_data_animals*. The SQL code is similar to that used for GPS data:

```
INSERT INTO main.activity_data_animals (
  animals_id,
  activity_sensors_id,
  acquisition_time,
  activity_x,
  activity_y,
  temp)
SELECT
  activity_sensors_animals.animals_id,
  activity_sensors_animals.activity_sensors_id,
  activity_data.acquisition_time,
  activity_data.activity_x,
  activity_data.activity_y,
  activity_data.temp
FROM
  main.activity_sensors_animals,
  main.activity_data,
  main.activity_sensors
WHERE
  activity_data.activity_sensors_code =
  activity_sensors. activity_sensors_code AND
  activity_sensors.activity_sensors_id =
  activity_sensors_animals.activity_sensors_id AND
  activity_data.acquisition_time >= activity_sensors_animals.start_time AND
  activity_data.acquisition_time <= activity_sensors_animals.end_time;
```

Exploring Activity Data and Associating with GPS Positions

Now, acceleration data are stored in the database and are ready to be analysed. If you explore the data set, you will notice that data are affected by many errors. Activity data error handling requires complex procedures. You could follow an approach similar to that used for GPS data, adding a validity code field and then running function to detect and tag outliers. For example, during pre-analysis data exploration, one should first look for activity data outside the measurement range (in our case greater than 255) and outside the time range during which the activity device was carried by the animal (this error should be filtered by the information on the time range of sensor deployment). These 'impossible' data will have to be excluded from the analysis. In addition, you should look for large data gaps (greater than four measurement intervals) within the data set. The frequency of occurrence and the length of gaps in the data set should be considered in the data analysis (some studies are able to cope with data gaps, and some studies need an interpolation to estimate missing measurements or even the exclusion of the whole data set). Faulty activity sensors can produce many or large data gaps. These devices should be recognised and replaced as soon as possible. We are not going

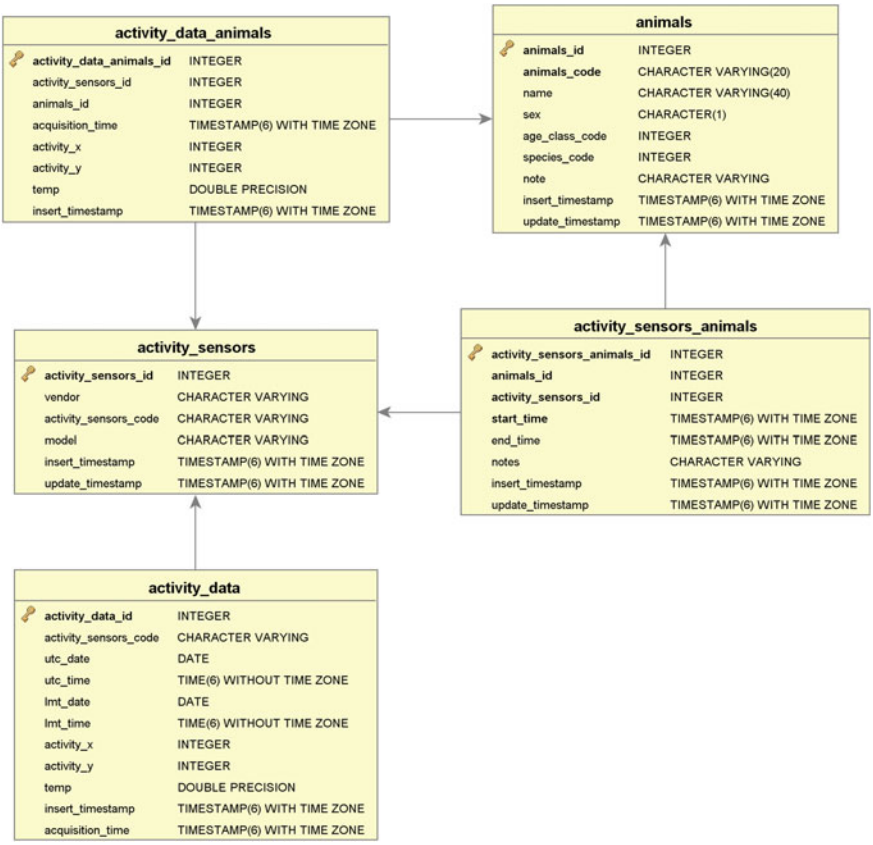

Fig. 12.1 Tables related to activity data management

into the details of these operations of quality check and outlier filtering because they are out of the scope of this guide.

You can now join activity and GPS data. In our activity data set, activity data are recorded with a frequency of 5 min, thus much more often than GPS positions (with a frequency of 4 h). Hence, you need to combine the two data sets in a way that makes sense for the specific scientific question to be asked. The most simple approach is to sub-sample the activity data to the same temporal resolution as the GPS positions. This can be done by a simple *JOIN* between the GPS data and the activity data, where you combine the data from the two sensors on the same animal at the same time. The result is that you associate the activity data record (what the animal is doing) with each GPS position (where the animal is doing it). You look for the activity data that are in a 5-min interval (2 min and 30 s after and before) from the GPS position acquisition time. The option *LIMIT 10* limits the result to

the first 10 records. You can see that with large amounts of data, the processing time can increase considerably:

```
SELECT
  gps_data_animals.animals_id AS id,
  gps_data_animals.acquisition_time AS acquisition_time_gps,
  activity_data_animals.acquisition_time::time AS time_act,
  ST_X(gps_data_animals.geom)::numeric(7,5) AS gps_x,
  ST_Y(gps_data_animals.geom)::numeric(7,5) AS gps_y,
  activity_data_animals.activity_x AS act_x,
  activity_data_animals.activity_y AS act_y
FROM
  main.gps_data_animals
INNER JOIN
  main.activity_data_animals
ON
  gps_data_animals.animals_id = activity_data_animals.animals_id AND
  ((gps_data_animals.acquisition_time -
activity_data_animals.acquisition_time) < interval '150 second') AND
  ((gps_data_animals.acquisition_time -
activity_data_animals.acquisition_time) > interval '-150 second') AND
  gps_validity_code = 1
LIMIT 10
```

The result is

```
 id |   acquisition_time_gps   | time_act |   gps_x    |   gps_y   | act_x | act_y
----+--------------------------+----------+------------+-----------+-------+---- ---
  3 | 2005-10-24 00:00:55+00 | 00:00:00 | 11.05471   | 46.00606 |    17 | 40
  3 | 2005-10-24 04:00:55+00 | 04:00:00 | 11.05290   | 46.00842 |     8 | 36
  3 | 2005-10-24 16:02:57+00 | 16:05:00 | 11.06117   | 46.00654 |    10 | 39
  3 | 2005-10-25 00:01:23+00 | 00:00:00 | 11.05217   | 46.00757 |    23 | 36
  3 | 2005-10-25 04:00:53+00 | 04:00:00 | 11.05288   | 46.00767 |    27 | 38
  3 | 2005-10-25 08:01:10+00 | 08:00:00 | 11.05778   | 46.00701 |     0 | 0
  3 | 2005-10-25 12:01:26+00 | 12:00:00 | 11.05861   | 46.00666 |     0 | 1
  3 | 2005-10-25 16:02:29+00 | 16:00:00 | 11.05875   | 46.00760 |    31 | 51
  3 | 2005-10-25 20:01:41+00 | 20:00:00 | 11.05478   | 46.00780 |    32 | 42
  3 | 2005-10-26 00:01:39+00 | 00:00:00 | 11.05620   | 46.00789 |     0 | 0
```

In the next query, you retrieve the average and standard deviation value for activity in the different land cover types. In this way, you can see whether the data indicate an obvious effect of the environment on the activity of the animal:

```
SELECT
  label3 AS land_cover,
  avg(activity_data_animals.activity_x)::numeric(5,2) AS avg_x,
  stddev(activity_data_animals.activity_x)::numeric(5,2) AS stddev_x,
  avg(activity_data_animals.activity_y)::numeric(5,2) AS avg_y,
  stddev(activity_data_animals.activity_y)::numeric(5,2) AS stddev_y,
  count(label3) AS num
FROM
  main.gps_data_animals,
  main.activity_data_animals,
  env_data.corine_land_cover_legend
WHERE
  gps_data_animals.animals_id = activity_data_animals.animals_id AND
  ((gps_data_animals.acquisition_time -
   activity_data_animals.acquisition_time) < interval '150 second') AND
  ((gps_data_animals.acquisition_time -
   activity_data_animals.acquisition_time) > interval '-150 second') AND
  gps_validity_code = 1 AND
  gps_data_animals.corine_land_cover_code =  corine_land_cover_legend.grid_code
GROUP BY
  label3;
```

The result reported is

```
        land_cover         | avg_x | stddev_x | avg_y | stddev_y | num
---------------------------+-------+----------+-------+----------+-----
 Mixed forest              | 18.05 |    32.87 | 22.72 |    32.18 | 378
 Broad-leaved forest       | 13.08 |    20.27 | 22.07 |    25.43 | 431
 Sparsely vegetated areas  | 13.82 |    12.44 | 29.44 |    23.44 | 101
 Coniferous forest         |  1.00 |          |  1.00 |          |   1
 Pastures                  | 29.81 |    37.71 | 36.74 |    35.38 | 772
```

As you can see, the highest activity rate is recorded in pasture lands, which might lead to interesting considerations on the behaviour of the animal.

In the last example, you retrieve the average activity value of the 6 activity records that are closest in time to each GPS position, which means 15 min after and before.

```
SELECT
  gps_data_animals.gps_data_animals_id AS gps_id,
  gps_data_animals.animals_id AS animal,
  gps_data_animals.acquisition_time,
  avg(activity_data_animals.activity_x)::numeric(5,2) AS avg_act_x,
  avg(activity_data_animals.activity_y)::numeric(5,2) AS avg_act_y
```

```
FROM
  main.gps_data_animals
INNER JOIN
  main.activity_data_animals
ON
  gps_data_animals.animals_id = activity_data_animals.animals_id AND
  ((gps_data_animals.acquisition_time -
  activity_data_animals.acquisition_time) < interval '15 minute') AND
  ((gps_data_animals.acquisition_time -
  activity_data_animals.acquisition_time) > interval '-15 minute') AND
  gps_validity_code = 1
GROUP BY
  gps_data_animals.gps_data_animals_id,
  gps_data_animals.animals_id,
  gps_data_animals.acquisition_time
ORDER BY
  gps_data_animals.acquisition_time
LIMIT 10;
```

The result is

```
 gps_id | animal |    acquisition_time    | avg_act_x | avg_act_y
--------+--------+------------------------+-----------+-----------
  15275 |      3 | 2005-10-23 20:00:53+00 |     27.67 |     56.33
  15276 |      3 | 2005-10-24 00:00:55+00 |     34.17 |     66.00
  15277 |      3 | 2005-10-24 04:00:55+00 |     16.00 |     33.83
  15280 |      3 | 2005-10-24 16:02:57+00 |      5.17 |     16.33
  15281 |      3 | 2005-10-24 20:01:49+00 |     15.83 |     29.83
  15282 |      3 | 2005-10-25 00:01:23+00 |     22.67 |     37.00
  15283 |      3 | 2005-10-25 04:00:53+00 |     58.50 |     55.33
  15284 |      3 | 2005-10-25 08:01:10+00 |      0.17 |      4.17
  15285 |      3 | 2005-10-25 12:01:26+00 |      0.00 |      1.83
  15286 |      3 | 2005-10-25 16:02:29+00 |     18.00 |     35.40
```

References

Berger A, Scheibe KM, Michaelis S, Streich J (2003) Evaluation of living conditions of free ranging animals by automated chronobiological analysis of behavior. BRMIC 35(3):458–466. doi:10.3758/BF03195524

Fröhlich M, Berger A, Kramer-Schadt S, Heckmann I, Martins Q (2012) Complementing GPS cluster analysis with activity data for studies of leopard (*Panthera pardus*) diet. S Afr J Wild Res 42(2):104–110. doi:10.3957/056.042.0208

Kranstauber B, Cameron A, Weinzerl R, Fountain T, Tilak S, Wikelski M, Kays R (2011) The Movebank data model for animal tracking. Environ Model Soft 26:834–835. doi:10.1016/j.envsoft.2010.12.005

Krop-Benesch A, Berger A, Hofer H, Heurich M (2013) Long-term measurement of roe deer (*Capreolus capreolus*) (Mammalia: *Cervidae*) activity using two-axis accelerometers in GPS-collars. Ital J Zool 80(1):69–81. doi:10.1080/11250003.2012.725777

Löttker P, Rummel A, Traube M, Stache A, Šustr P, Müller J, Heurich M (2009) New possibilities of observing animal behaviour from a distance using activity sensors in GPS-collars: an attempt

to calibrate remotely collected activity data with direct behavioural observations in red deer *Cervus elaphus*. Wild Biol 15(4):425–434. doi:10.2981/08-014

Qasem L, Cardew A, Wilson A, Griffiths I, Halsey LG et al (2012) Tri-axial dynamic acceleration as a proxy for animal energy expenditure; should we be summing values or calculating the vector? PLoS ONE 7(2):e31187. doi:10.1371/journal.pone.0031187

Reimoser S (2012) Influence of anthropogenic disturbance on activity, behaviour and heart rate of roe deer (*Capreolus capreolus*) and red deer (*Cervus elaphus*), in context of their daily and yearly patterns. In: Cahler AA (ed) Deer: habitat, behaviour and conservation. Nova Science Publishers Inc., Hauppauge, pp. 1–95. ISBN: 978-1-62100-676-3

Scheibe KM, Gromann C (2006) Application testing of a new three-dimensional acceleration measuring system with wireless data transfer (WAS) for behavior analysis. Behav Res Meth 38(3):427–433. doi:10.3758/BF03192796

Shepard ELC, Wilson RP, Quintana F, Laich AG, Liebsch N, Albareda DA, Halsey LG, Gleiss A, Morgan DT, Myers AR, Newman C, Macdonald DW (2008) Identification of animal movement patterns using tri-axial accelerometry. Endanger Species Res 10:47–60. doi:10.3354/esr00084

Chapter 13
A Bigger Picture: Data Standards, Interoperability and Data Sharing

Sarah Cain Davidson

Abstract Data sharing is of growing interest in science and in ecology. Many research questions in ecology, particularly those addressing global change, require large, long-term data sets that cannot be collected by any one research group alone. Moreover, an increasing number of funding providers and publishers require that researchers make their data available in some form to other researchers or to the public. Benefits to sharing your data can include new collaborations and publications, increased citations of your research, expansion of successful wildlife management strategies to new areas or species, and fulfillment of journal and funding requirements for data sharing and management plans. As you develop your database, it is worth considering ways to share your data, either with specific collaborators or with the public, and to at minimum make a description of your data set publicly available. And, as we have emphasised throughout this book, the data organisation and documentation required for sharing data should be a standard part of data collection regardless of the end uses of your data. The goal of this chapter is to introduce you to existing ecological data standards and a variety of ways to make your database archivable and usable for additional analyses.

Keywords Data management · Data sharing · Data standards · Metadata

S. C. Davidson (✉)
Max Planck Institute for Ornithology, Am Obstberg 1, 78315 Radolfzell, Germany and
Department of Civil, Environmental and Geodetic Engineering, The Ohio State University,
2070 Neil Ave, Columbus, OH 43210, USA
e-mail: sdavidson@orn.mpg.de

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_13,

Introduction

Data sharing is of growing interest in science and in ecology (Vision 2010; Reichman et al. 2011). Many research questions in ecology, particularly those addressing global change, require large, long-term data sets that cannot be collected by any one research group alone (Wolkovich et al. 2012). Therefore, scientists may wish to combine some or all of their data with those of collaborators, or with archived data collected in the past, to complete a new analysis. Moreover, an increasing number of funding providers and publishers require that researchers make their data available in some form to other researchers or to the public. Lastly, when designing and managing a database, it is important to remember that the specific study for which the database was created will one day be finished, and those who developed it will no longer be spending their days (and nights) thinking about all the details of the study. In order for a database to remain meaningful for possible future use, it is important to consider possibilities for long-term data preservation.

Benefits to sharing your data can include new collaborations and publications (Lacher et al. 2012), increased citations of your research (Piwowar et al. 2007), expansion of successful wildlife management strategies to new areas or species, and fulfillment of journal and funding requirements for data sharing and management plans (Whitlock 2011). In addition, it is our responsibility as scientists to promote new knowledge by making data available, as appropriate, to the rest of the scientific community and even to the public, who fund many wildlife tracking programmes and research studies.

As you develop your database, it is worth considering ways to share your data, either with specific collaborators or with the public, and to at minimum make a description of your data set publicly available. The goal of this chapter is to introduce you existing ecological data standards and ways to make your database archivable and usable for additional analyses.

Although shared data from many other fields, such as hydrology, meteorology and genetics, have been widely used for many years, many wildlife tracking researchers remain reluctant to share data. Common concerns are that data will be misunderstood or used without proper acknowledgment, that sensitive data will get into the wrong hands or that they do not have enough time or resources to properly share data. These concerns highlight the need for appropriate methods for sharing data, combined with good data management and thorough documentation. As is described in this chapter, existing methods for sharing data address each of these concerns, from enabling data citation, to limiting sharing to trusted users, to encouraging communication between data owners and users, to providing free tools and support. And, as we have emphasised throughout this book, data organisation and documentation is not only needed when data will be shared with others, but also should be a standard part of data collection regardless of the end uses of your data.

An essential component of data sharing is the use of standards. Given the heterogeneity of methods, data sets and software used in the field of wildlife tracking, combined with the potential benefits of collaboration, there is a need for internationally recognised standards for describing and sharing data. Several examples of existing standards are described later in this chapter. These standards can help to ensure compatibility between different software platforms, research groups and databases. In addition, data standards play an important role in improving data quality and can liberate data from the specific aim for which they were collected. Adhering to such standards ensures that data can be reused for a wide range of purposes, maximising the returns of research funding and facilitating multi-species, large-scale and long-term ecological studies.

Describing Data

In order for raw data files to be understandable to others, they need to be well described. The meaning and format of each term used in your database should be defined, including the following:

- terms describing the actual data set attributes, e.g. the reference coordinate system of locations, timestamp format, units and precision;
- terms describing entities like sensors and animals, such as sex, serial number or species name; and
- terms describing the entire database or discrete subsets of it, such as the title, authors, keywords, time and geographic range of the data set used in a particular analysis.

There are several general rules to follow when describing terms in a data set:

- Use controlled vocabularies (a set of predefined words or terms) where possible. For example, if you are classifying migration stage for each record in your data set, allow only a discrete list of terms, such as ‘stopover’, ‘northward migration’ and ‘breeding grounds’. This supports consistent classification, prevents spelling errors and allows for easier analysis. Database tools such as lookup tables and constraints can be helpful in implementing these vocabularies.
- Never use a term twice if the definition is not exactly the same. If you are using two types of sensors, label them ‘GPS sensor’ and ‘activity sensor’, for example, rather than calling them both ‘sensor’ and risking confusion or errors, even if it is clarified by contextual information (e.g. the name of the table or of the schema where the information is stored).
- Where possible, data values should follow common standards—for example, providing timestamps in Coordinated Universal Time (UTC) and using species names from a published taxonomy.

- If you use codes or ambiguous shortened names (e.g. 'CC' for '*Capreolus capreolus*'), be sure to include tables that provide a full translation of codes used and that these tables are always included with any data transfer.
- To the extent possible, rely on standard database design approaches. Consider using the data and metadata standards described below, and follow generic table formats like those we present in this guidebook. If data structure and definition are unnecessarily complex or specific to the original context for which a database was developed, merging data sets and linking your data to external analysis tools becomes more complicated.
- Most importantly, make sure to maintain a written definition of all terms in your database that is available to all users. The definition should have a text description of the term along with any units, valid ranges, example values or controlled lists. The written definitions should explain where the values come from, such as the source of altitude estimates that may come from a DEM or from the GPS unit, or the method for determining habitat or behaviour. You could create this as a separate table in the database or as a plain text file.

Data and Metadata Standards

Several standards or schemas have been designed to deal specifically with describing ecological and geospatial data. These standards support description, discovery and integration of biological and geospatial data and are used by a wide range of research institutes, universities, museums, government agencies and other organisations. Standards provide relevant terms and definitions, have policies governing how to maintain and use the terms and document the history of changes to the standard. Where possible, it may be helpful to use terms from one or more of these standards in your database. This allows you to use and reference existing definitions, rather than writing your own, and would make it easier to share your data or metadata with databases such as the Global Biodiversity Information Facility (GBIF[1]) or the earth observation database DataONE[2].

Note that the difference between 'data' and 'metadata' is not clearly defined and will vary depending on the context. In general, metadata refers to 'data about data' or information that describes a data set. For example, descriptions of study animals might be considered 'metadata' describing your tracking data in one context, while in another, this information might be a part of your data set, with 'metadata' referring to a description of the entire study (such as title, authors and the time period of data collection). For our purposes, it may be helpful to think of metadata standards as useful for *finding* data and data standards as useful for *integrating or combining* data.

[1] http://www.gbif.org/.

[2] http://www.dataone.org/.

Three metadata standards specific to biology are Darwin Core, Access to Biological Collections Data (ABCD) and Ecological Metadata Language (EML). These standards are currently in use around the world and are freely available. In addition, they have support for geographic and temporal information.

The **Darwin Core** and **ABCD** standards are developed by Biodiversity Information Standards or TDWG (formerly the Taxonomic Database Working Group). Initially developed for use with natural history collections, **Darwin Core**[3] is widely used and includes terms for describing species occurrence data, including physical specimens, observations and digital records. It is focused primarily on terms that are generically applicable to natural history collections[4].

The **ABCD** standard[5] supports species occurrence data and includes around 1,200 terms (they refer to these as 'concepts'). It includes a larger number of terms and a more complex structure than Darwin Core, making it able to describe data and relationships between them more thoroughly, but requiring more technical expertise to fully implement (Wieczorek et al. 2012)[6].

EML[7] is a metadata standard developed by the Knowledge Network for Biocomplexity for describing ecological data (Higgins et al. 2002; Fegraus et al. 2005). It is open source and implemented by voluntary project members. It was designed primarily to describe data sets and other digital resources. EML consists of several modules that can be adopted by users as needed, including modules to support detailed descriptions of methods, attributes, tables in relational databases, and raster and vector geographic information[8].

In addition, the following other data and metadata standards may be useful:

The **Content Standard for Digital Geospatial Metadata** (**CSDGM**[9]), developed by the US Federal Geographic Data Committee (FGDC), is the current Federal metadata standard in the USA for geospatial data. This standard has a Biological Data Profile to provide additional support for biological data. Although it has been widely used throughout the USA, the FGDC now supports the transition to the ISO 19115 standard (see below).

The **ISO 19115 standard** is the International Organization for Standardization's (ISO[10]) metadata standard for describing geographic data. Unlike the other resources listed here, ISO standards are not freely available (i.e. must be paid for).

---

[3] http://rs.tdwg.org/dwc.

[4] Terms are currently described at http://rs.tdwg.org/dwc/terms/index.htm.

[5] http://www.tdwg.org/activities/abcd.

[6] Terms are currently described at http://wiki.tdwg.org/twiki/bin/view/ABCD/AbcdConcepts.

[7] http://knb.ecoinformatics.org/software/eml.

[8] Terms and modules are described at http://knb.ecoinformatics.org/software/eml/eml-2.1.1/index.html.

[9] http://www.fgdc.gov/metadata/geospatial-metadata-standards#csdgm, http://www.fgdc.gov/metadata/csdgm/.

[10] http://www.iso.org.

The **Open Geospatial Consortium** (OGC[11]) provides publicly available interoperability standards (see below). While these are not specifically data or metadata standards, some of their standards do include relevant data/metadata terms and schema that are used to implement interoperability.

In addition to these resources, manufacturers of tagging equipment typically provide data using fairly standardised formats and data attributes. Simply following the format delivered by popular manufacturers or other data providers, such as Argos, may be a simple way to make large amounts of data easy to share and compile. However, keep in mind that formats, attributes and units differ between manufacturers and sensors and that these companies are in the business of selling equipment, not maintaining databases. Combining data from different manufacturers and sensor types, each with their own specialised terms and data structure, can require significant effort. In addition, you will find that manufacturer-provided data formats often change over time and can be ambiguous, resulting in data files that misleadingly appear to be in the same format—for example, in some cases, users are allowed to choose a time zone in which data are delivered, although this choice is not indicated anywhere in the data file.

Lastly, you may want to look at data formats used by existing online animal tracking databases, such as those listed at the end of this chapter, in particular if you intend to use one of these databases for sharing or analysis. For example, see the Movebank Attribute Dictionary[12]. While these do not constitute official standards, the managers of these databases have developed data formats that are shared by large groups of data owners and could be extended or modified to meet your specific requirements.

Interoperability

Data that use shared metadata and data standards may be stored in diverse data formats and in online and offline databases. To allow others to locate and access metadata in a shared format, there must be ways to search metadata for species name, location, time period of data collection, etc., using Web-based databases or search engines. To actually combine multiple data sets for analysis requires additional work and may be time-consuming or impossible to do manually. In order to properly and efficiently search for or integrate data sets, we require interoperability.

In this context, interoperability can be generally defined as the ability for multiple databases, analysis software or other relevant systems to work together.

[11] http://www.opengeospatial.org/.

[12] https://www.movebank.org/node/2381.

For example, consider the interoperability of your database with an external client software for analysis:

- Not interoperable: Your data are stored in a proprietary format that cannot be read by the analysis software, and there is no easy way to export the data for use with the software.
- Somewhat interoperable: You can easily export all or part of your database as a .csv file, which can be read by the software.
- Very interoperable: You can query your database directly from the software, run an analysis and automatically store the results in your database.

There are several general ways in which you can make your database more interoperable with other systems. For example, you may use database software like PostgreSQL and PostGIS, which are open source, widely support international standards, and are likely to be maintained in the future. Using common non-proprietary file formats such as .csv rather than .xlsx when needed will minimise the chances of files becoming unreadable by contemporary software. Lastly, using established data and metadata standards such as those described above will make it more likely that your data can be understood and integrated with other data sets and software in the future.

Full interoperability with software tools, other databases and search engines requires implementation of more specific technical standards, which is beyond the scope of this guide. These include specifications for exchanging information using data/metadata exchange file formats, such as Extensible Markup Language (XML) or Resource Description Framework (RDF), and transfer protocols, such as TAPIR[13]. These standards are necessary, for example, to allow computers to automatically read data or metadata, retrieve search results, and present them in a way understandable to the user.

The **OGC**[14] is an international consortium that develops voluntary standards for interoperability of GIS data. PostGIS, used in this guide, follows the OGC's 'Simple Feature Access—Part 2: SQL Option' specification[15] and has been certified compliant with the 'Simple Features—SQL—Types and Functions 1.1' specification.

Publish Your Metadata

An alternative to publishing a data set in full is to make a description, in the form of metadata about your data set, available to a wider community. This makes it easier for other researchers to find out about your research and contact you about

[13] http://www.tdwg.org/activities/tapir/specification.

[14] http://www.opengeospatial.org/.

[15] http://www.opengeospatial.org/standards/sfs.

possible collaborations. Databases with searchable study metadata often exist at the level of the research group, university, region or country. To be broadly useful, these typically require only a minimal number of descriptive terms that are applicable to all studies in the database.

To reach the widest possible audience, here are two global online databases where metadata about your wildlife tracking database could be shared:

- **GBIF** (**Global Biodiversity Information Facility**) publishes metadata about primary biodiversity occurrence data.[16] A list of contacts is available at http://www.gbif.org/communications/directory-of-contacts/regional-nodes
- **DataONE** (**Data Observation Network for Earth**) publishes metadata about earth observational data[17]. A list of member nodes is available at www.dataone.org/current-member-nodes

To contribute, you must be associated with a member node. Member nodes may include research institutes, government agencies and other organisations. In the case of GBIF, most countries also have a national node, and so if you are not affiliated with an existing member organisation, you could contact your national node to find out how to get involved.

It is relatively easy to store metadata within your PostgreSQL database. In addition to storing definitions for each term in the database (see **Describing data** above), you can create descriptive metadata for the database itself and for subsets within it, such as the set of records used for a specific publication or analysis. These metadata can include required terms for external databases and can be stored within your database in an XML format that complies with XML schema for the metadata standards described above using the XML data type in PostgreSQL.

In addition, many software programs exist to help researchers write and publish metadata in interoperable formats. One example is **Morpho**[18], a free, user-friendly software tool developed to help researchers write and publish metadata without special knowledge of technical interoperability requirements. Morpho allows you to write detailed metadata about ecological data sets and individual data tables that comply with the EML standard in XML format. After creating metadata, you can store files locally or upload the metadata, and even data tables, to the Knowledge Network for Biocomplexity, where they are searchable and available to other registered members (it is possible to restrict access to specific collaborators). This program is available from the Knowledge Network for Biocomplexity[19] (Higgins et al. 2002).

[16] http://www.gbif.org.

[17] http://www.dataone.org.

[18] https://knb.ecoinformatics.org/morphoportal.jsp.

[19] https://knb.ecoinformatics.org.

Publish and Share Your Data

Publishing your data allows you to share them in a more formal and structured way than exchanging files individually with collaborators and can make it accessible to a much wider research community. It also makes it easier for others to properly cite your data and allows you to list these citations in your CV as valuable research products in their own right. Depending on how you publish your data, you can make them available to the public or to specific user groups, define explicit terms of use, and ensure that some or all of a data set is permanently archived and remains accessible. Publishing data commonly involves the following:

- a review process to ensure the quality of data and related documentation;
- assignment of a persistent identifier such as a Digital Object Identifier (DOI) or Life Science Identifier (LSID) to ensure that the item will remain permanently available; and
- data licences that provide explicit conditions for reuse, such as those offered by Creative Commons[20] and Open Data Commons[21, 22].

If you have a completed data set that you would like to make available to the public and scientific community, you can submit it for review and publication. Several journals publish 'data papers', which include a biological data set along with a written description of the data, for research in biology and ecology. These include **Biodiversity Data Journal**, **Dataset Papers in Science**, **Ecological Archives**, **and Scientific Data**. In addition, there are databases that publish 'data packages' or sets of non-proprietary files associated with a written publication (see Penev et al. 2011). These include **Data Dryad**[23], which publishes data sets in the life sciences, and the **Movebank Data Repository**[24], which publishes animal tracking data in a standardised format.

Share Your Data Without Publishing

In some cases, the formality and permanence of publishing your data as described will not be the best option. For example, publicly revealing precise breeding or foraging locations of endangered populations may put them at risk. More commonly,

---

[20] http://www.creativecommons.org.

[21] http://opendatacommons.org.

[22] Note that making data files available as supplementary material along with a written article in general does not fit this definition of publication. In most cases the files are not part of the review process, and there is no guarantee by the publisher that the files will remain available (Anderson et al. 2006).

[23] http://www.datadryad.org.

[24] http://datarepository.movebank.org.

Table 13.1 A summary of several online databases that host animal tracking data

Database name	Description	Data access	Host
Australian Animal Tagging and Monitoring System (http://imos.aodn.org.au/webportal)	Part of the Integrated Marine Observing System, which includes other types of observational oceanic data; animal tracking data include marine animals from Australia to Antarctica	All data are available to the public	The Australian Government
Eurodeer (European Roe DEER Information System) (www.eurodeer.org)	A database used and maintained by researchers studying European roe deer (*Capreolus capreolus*) to support collaborative analysis	Data are shared between participating research groups; access can be requested, subject to agreement with 'terms of use'	Fondazione Edmund Mach
Global Procellariiform Tracking Database (www.seabirdtracking.org)	A compilation of existing seabird tracking data, obtained through personal communication with data owners to support the development of distribution maps and conservation recommendations	Controlled by data owners, with some summary information visible to the public	BirdLife International
Global Tagging of Pelagic Predators (www.gtopp.org)	A compilation of marine tracking and related bio-logged data	Select data are available to the public	An international collaboration of government and research institutions and equipment manufacturers
Movebank (www.movebank.org)	A data storage, sharing, analysis, and archiving platform for animal movement and related sensor data (the Movebank Data Repository)	Controlled by data owners; can be shared with select users or the public	The Max Planck Institute for Ornithology and the University of Konstanz
OBIS-SEAMAP (seamap.env.duke.edu)	A component of the Ocean Biogeographic Information System that allows the public to search for and aggregate information about marine species distributions, including marine mammals, seabirds, and sea turtles	All data are available to the public	Duke University

(continued)

Table 13.1 (continued)

Database name	Description	Data access	Host
Seaturtle and WildlifeTracking (www.seaturtle.org, www.wildlifetracking.org)	A data storage, sharing, and analysis platform for data collected using the Argos satellite system	Controlled by data owners	A private 501(c)3 tax-exempt organisation in the US
Wireless Remote Animal Monitoring (WRAM) (www.slu.se/WRAM)	A data storage, networking, sharing, and analysis platform for movement and other sensor data from fish and wildlife	Controlled by data owners; can be shared between participating research groups	The Swedish University of Agricultural Sciences

when research is ongoing and results of analysis are unpublished, you will likely want to share data only with specific people, such as collaborators or funding providers. Data sharing for the purpose of ongoing collaborative analysis requires different tools. These tools should allow you to define access rights to specific people, allowing them to view and/or add and edit data, and provide an infrastructure that helps multiple researchers to put their data into the same format.

In Chap. 2 and elsewhere in this guide, we have briefly described how you can use PostgreSQL to define multiple users and distinct access rights for each within your database[25] and allow remote connections. These features can support data sharing with colleagues, along with all the analysis and database design options PostgreSQL provides.

In addition, there are many existing online databases for sharing animal tracking data. One or more of these databases could provide a useful resource to complement your personal research database. These databases have varying Data-sharing options and are available to different sets of users and study types. They provide a way for a wide range of researchers, educators and conservation groups to find out about your research. Some of these databases also provide data-sharing and data-collection services, as well as a visualisation and analysis tools, that could be particularly useful for those with limited technical facilities. Brief descriptions of several of these are given in Table 13.1.

References

Anderson NR, Tarczy-Hornoch P, Bumgarner RE (2006) On the persistence of supplementary resources in biomedical publications. BMC Bioinf 7:260. doi:10.1186/1471-2105-7-260

Fegraus EH, Andelman S, Jones MB, Schildhauer M (2005) Maximizing the value of ecological data with structured metadata—an introduction to Ecological Metadata Language (EML) and principles for metadata creation. Bull Ecol Soc Am 86(3):158–168. doi:10.1890/0012-9623(2005)86[158:MTVOED]2.0.CO;2

Higgins D, Berkley C, Jones MB (2002) Managing heterogeneous ecological data using Morpho. In: Proceedings of the 14th international conference on scientific and statistical database management, pp 69–76. doi:10.1109/SSDM.2002.1029707

Lacher TE, Boitani L, da Fonseca GAB (2012) The IUCN global assessments—partnerships, collaboration and data sharing for biodiversity science and policy. Conserv Lett 5:327–333. doi:10.1111/j.1755-263X.2012.00249.x

Penev L, Mietchen D, Chavan V, Hagedorn G, Remsen D, Smith V, Shotton D (2011) Pensoft data publishing policies and guidelines for biodiversity data. Pensoft Publishers. www.pensoft.net/J_FILES/Pensoft_Data_Publishing_Policies_and_Guidelines.pdf

Piwowar HA, Day RS, Fridsma DB (2007) Sharing detailed research data is associated with increased citation rate. PLoS ONE 2(3):e308. doi:10.1371/journal.pone.0000308

Reichman OJ, Jones MB, Schildhauer MP (2011) Challenges and opportunities of open data in ecology. Science 331(703):703–705. doi:10.1126/science.1197962

Vision TJ (2010) Open data and the social contract of scientific publishing. Bioscience 60(5):330–331. doi:10.1525/bio.2010.60.5.2

[25] http://www.postgresql.org/docs/9.2/static/user-manag.html.

Whitlock MC (2011) Data archiving in ecology and evolution—Best practices. Trends Ecol Evol 26(2):61–65. doi:10.1016/j.tree.2010.11.006

Wieczorek J, Bloom D, Guralnick R, Blum S, Döring M, Giovanni R, Roberson T, Vieglais D (2012) Darwin Core—An evolving community-developed biodiversity data standard. PLoS ONE 7(1):e29715. doi:10.1371/journal.pone.0029715

Wolkovich EM, Regetz J, O'Connor MI (2012) Advances in global change research require open science by individual researchers. Glob Change Biol 18:2102–2110. doi:10.1111/j.1365-2486.2012.02693.x