Tutorial to the RObsDat-package

Dominik Reusser

February 27, 2015

1 Introduction

Research data is a valuable asset. As Gold [2007] put it:

> "[. . .] data is the currency of science, even if publications are still the currency of tenure. To be able to exchange data, communicate it, mine it, reuse it, and review it is essential to scientific productivity, collaboration, and to discovery itself."

This is also true for hydrological data. However, making data exchangeable and communication requires a well organized data management, for which funding is often not or only to a limited extend available. Efficient methods to manage hydrological data for small and medium scale projects has been missing up to now. What is required is a system that

- stores data in a structured (adhering to standards) and efficient way
- allows centralized or decentralized data management
- is simple to set up with limited or intermediate database skills
- avoids date time errors due to wrong conversion
- provides flexible and efficient ways of data import and data export
- supports and documents data cleaning steps
- provides tools for data analysis
- runs independent of the operating system
- allows inspection of the code for debugging and further development
- provides documentation

The RObsDat package is designed to fill this gap in conjunction with the R environment and is currently in a testing phase where code is made more efficient and more stable. In order to use RObsDat, you need to understand the format used for data storage. This will be presented in section 2. The role sharing between the R environment and the package is briefly touched upon in section 3. Different database engines are supported by the package and basics are presented in section 4. You might decide to initially skip sections 3 and 4 and use the simple example presented in section 5 to learn the practical usage of RObsDat. We conclude (section 6) with a short outlook and an invitation to contribute.

2 Observations Data Model

Different database designs may be used to manage hydrological (and other) data. RObsDat is designed to store point observations which come with the following properties. To store data with a stronger spatial component (such as satellite images and derivatives there off), a (spatio-temporal) GIS system is recommended.

In general, hydrological observations come as a triple including time, variables and space. This triple is usually extended by information about data quality and the data source/collection method. CUAHSI (The Consortium of Universities for the Advancement of Hydrologic Science) has presented a specification for the Observations Data Model (ODM) which is currently available as version 1.1 [Tarboton et al., 2008]. This data model is implemented by RObsDat, however the package has been designed to be potentially compatible with other data models. In addition, CUAHSI recommends to use a controlled vocabulary (CV) for the most important attributes to enhance semantic consistency and provides these through a web service. RObsDat is able to import these CV.

Before you can store data with RObsDat, you need to supply meta-data details for the location, the variable, the quality and the source using the commands `addSite`, `addVariable`, `addQualityControlLevel`, and `addSource`. All of these meta-data sets is stored in a separate data table with additional links to specific CV-tables. See the example in section 5 to get started. The full data model is presented in appendix A.

3 Role sharing between RObsDat, R and other packages

A part of the power of managing your data with RObsDat is the embedding environment R. It provides great flexibility for data import, scripting of standard processes, platform independence. RObsDat relies on these advantages of R and attempts to do the management of your data in database well. When you receive data from the database, use the power of R to do subsequent (statistical) analyses and to create plots. Important packages RObsDat depends on are related to the management of time-objects and space-time objects:

3.1 Date formats, zoo/xts

When adding data values, RObsDat expects you to provide time information in the `POSIXct` format. This is the interface between R and the POSIX standard for date/time handling. You may also store your data as time series data in form of a `zoo` or `xts` object. See the corresponding package documentations. You will need a basic understanding of these to work with RObsDat.

3.2 spacetime

This version of the package will return data as spacetime objects, which is a first step to provide possibilities for efficient handling and analysis of spatio-temporal data in R. See the introductory journal article to the package for more detail [Pebesma, 2012].

4 Database engines

RObsDat has been tested and made compatible with three different, open source database engines: SQLite, MySQL and PostgreSQL. The most simple set-up is by using SQLite, which stores all the information in a single file with the extension db. The default is a file in the current working directory and the name `RODM.db`. You can share the db-file, such that other users can also access your database. Free graphical user interfaces to SQLite databases exist for most operating systems and can be found on the web.

In case you don't know what to do, `getDefaultDB()` will install a preconfigured database file shipped with the package. This command also work if your db file is called `RODM.db` and is situated in your working directory.

```
> getDefaultDB()
```

To connect to a different SQLite database, you may use the following set of commands:

```
> require("RObsDat")
> require("RSQLite")
> m <- dbDriver("SQLite")
> dbname = "database.db"
> con <- dbConnect(m, dbname = dbname)
> sqhandler <-  new("odm1_1Ver", con=con)
> options(odm.handler=sqhandler)
```

However, concurrent data management by multiple users is not advisable with such a set-up. Running a database server is recommended in such a case. Contact your system administrator if you need to set-up a database server. To connect to a running server, you may use the following commands.

```
> #connect to postgreSQL database
> require("RObsDat")
> require("RPostgreSQL")
> m <- dbDriver("PostgreSQL")
> con <- dbConnect(m, user="a_user", password="secret", dbname="obsdat")
> sqhandler <-  new("odm1_1Ver", con=con)
> options(odm.handler=sqhandler)
> #connect to MySQL database
> require("RObsDat")
> require("RMySQL")
> m <- dbDriver("MySQL")
> con <- dbConnect(m, user="a_user", password="secret", dbname="obsdat")
> sqhandler <-  new("odm1_1Ver", con=con)
> options(odm.handler=sqhandler)
```

5 An example session

We start by obtaining a preconfigured database, shiped with the package as `RODM.db` into our current working directory. All modification will be stored in this file. If you again use `getDefaultDB()` in the same working directory, a connection will be made to the existing database file and you will have access to all the former modifications. If you want to start with a blank database, make sure to delete the file. However, RObsDat is designed to be smart about repeated execution of commands and will avoid duplication of records. Let's get started:

```
> require("RObsDat")
> getDefaultDB()
```

Before adding data values, we need to set up the meta-data. We will check what the correct names of the metadata are:

```
> #Store metadata in database
> getMetadata("SpatialReference", SRSName="WGS84", exact=TRUE)

  ID SRSID SRSName IsGeographic Notes
1  3  4326   WGS84         TRUE

> addSite(Code="testSpatialPoints", Name="Virtual test site", x=25, y=56,
+         LatLongDatum="WGS84", Elevation=350, State="Germany")

NULL

> getMetadata("Units", Name="degree celsius")

   ID                                      Name          Type  Abbreviation
1  94                            degree celsius   Temperature          degC
2 311                    squared degree Celsius   Temperature      (DegC)^2
3 313          meters per second degree Celsius   Temperature      m/s DegC
4 315 degree Celsius millimoles per cubic meter Concentration DegC mmol/m^3

> getMetadata("VariableName", Term="Temperature")

                            Term
1            Battery temperature
2         Temperature, dew point
3 Temperature, transducer signal
                                                 Definition
1 The battery temperature of a datalogger or sensing system
2                                     Dew point temperature
3                         Temperature, raw data from sensor

> addVariable(Name="Temperature, transducer signal", Unit="degree celsius", ValueType="Fie
+         GeneralCategory="Hydrology", Code="test_temp")
> addQualityControlLevel(ID=2,Code="test_ok", Definition="The default values")
> addISOMetadata(TopicCategory="Unknown", Title="Testdata",
+         Abstract="This data is created to test the functions of RObsDat")

NULL

> addSource(Organization="Your Org", SourceDescription="Madeup data",
+         SourceLink="RObsDat Documentation", ContactName="Yourself",
+         Metadata="Testdata")
```

With this set of commands, we are done preparing meta-data for a first variable and location. Note that the information about the source is stored in two tables, one including the Dublin Core information and the other remaining meta-data. From the example, it should be simple to add meta-data for other locations or other variables. For the tutorial, we are content with one site and one variable.

As a next step, we can start to add data. Let's create a tutorial data set and store it as xts. We introduce two data errors that we want to correct in a subsequent test.

```
> require(xts)
> example.data <- xts(1:40, seq(as.POSIXct("2014-01-01", tz="UTC"),
+                   as.POSIXct("2014-02-09", tz="UTC"), length.out=40))
> example.data[40] <- 30
> example.data[35] <- 22
```

With loggers, we often obtain multiple files with partially redundant data (the later dataset containing some of the information from a previous dataset). RObsDat is able to detect redundant data. As long as no conflicts arise, the package is silent about redundant entries. If present, existing conflicts are presented and the user is asked about the desired action. We simulate this by first importing the first 20 data sets, modifying our data and then importing the remaining data set.

```
> addDataValues(example.data[1:20], Site="Virtual test site", Variable="test_temp",
+         Source="Madeup", QualityControlLevel="test_ok")

Importing column  1 out of 1

>
```

By importing the remaining data through insert the hole values, only the new data will imported. If an existing value was changed you get the question what should happen with them.

```
> #Avoid duplicates autmatically
> example.data[15] <- 30
> addDataValues(example.data, Site="Virtual test site", Variable="test_temp",
+         Source="Madeup", QualityControlLevel="test_ok")

Importing column  1 out of 1
```

Note the "split responsibility" between the R environment and RObsDat (section 3). You will use the power and flexibility of R to get the data into the environment and convert dates into a POSIX-format, while the package provides support mechanisms to get the data in a consistent form into the database. Note that the import method is quite smart about data configurations and automatically detects whether meta-data is valid for columns or rows or is provided as full table (check this out yourself and report unexpected behaviour to the maintainer). Also, typos and multiple versions for site and variable names are interactively caught and the database remembers synonyms for data sets. Data in the database can be accessed and filtered in an intuitive way:

The data are restored in several STFDF. RObsDat contain a class with an inherited STFDF from the packages spacetime. STFDF is for spatio-temporal data with a full data frame. It shows n spatial points and m times. Foreach location and time exists observations. It deals with spatio-temporal data and provides special time series analysis. The object of class inherited_stfdf contains one stfdf with the main-data: spatial data of package sp and temporal informations of class xts. he actual data is stored in the form of a data.frame. Furthermore it contains two stfdf-objects with the 'ValueIDs' and the 'DerivedFromIDs' and also one data.frame with the Meta-informations.

Thus 'getDataValues' returns a spacetime object. RObsDat has powerful mechanism to retrieve the data needed through data base queries. The query can specify additional contraints (e.g. site):

```
> allData <- getDataValues()
> testSiteData <- getDataValues(Site="test")
```

Further you can access the slots of object by take advantage of the '@'. For example: testSiteData@Metadata testSiteData@sp testSiteData@time testSiteData@data

To retrieve the data through sub-setting the spacetime object is also efficient:

a[i, j, k] i= spatial features j = temporal instances k = data variable(s)

You can selected the data by using numerics and characters. In the following code line 'allData' is sub-setted to the second and third location, the first twenty dates and the temperature. To choose the data variable is especially interesting, when the selected data contains more than one variable. Advice: The variable must have the correct term with combination of the respective metadata-id. For choose the right term look at the column name of allData@data.

`selectedData = allData[2:3, 1:20, "Temp2erature, transducer signal\_1"]`

Furthermore you can miss out some parameters: `selectedData = allData[1:2,10:20] selectedData = allData[,10:20]`

Attention: if the dataset contains only one location (or it is subset only one point) it's impossible to get a inherited spacetime-object. So you get a object of xts. If only one time is selected you get a SpatialPointsDataFrame.

The inherited stfdf-data can visualized in different plots. For example:

stplot(selectedData, mode='tp', type = 's') stplot(selectedData, mode='xt')

When you add more variables you can visualized also plots with more than one variable or several locations.

Raw data often needs to be cleaned before it can be used. RObsDat supports this and allows to reconstruct data modification operations by use of version management. The following lines demonstrate this:

```
> #Version management
>
> testSiteData <- getDataValues(Site="test")
> to.correct <- which(testSiteData@data > 30)
> testSiteData@data[to.correct,] <- 20
> testSiteData@data[39,] <- 32
> #ToDo - doesn't pass test
> #updateDataValues(testSiteData, "Correction of wrong value")
```

```
>
> ver2 <- testSiteData
> ver2@data[10:13,] <- 60
> #updateDataValues(ver2, "Changing more data")
>
> testSiteData <- getDataValues(Site="test")
> ver3 <- testSiteData
> ver3@data[30:32,] <- 33
> #updateDataValues(ver3, "Ups, I used 60 instead of 33 by mistake")
>
```

And finally we want to remove values. At first only one value. In this case the 36'th value. Rules which are explained by selection come into effect. So you have to add '@ValueIDs' to remove only one value (one time point) or several values, when the dataset contains only one location. deleteDataValues(testSiteData@ValueIDs[,36], "Remove a value") deleteDataValues(testSiteData@ValueIDs[,10:14], "Remove several values")

If you want to delete by boolean, then you have to use the addition @ValueIDs, too. to.delete <- testSiteData@data == 60 if(any(to.delete)) deleteDataValues(testSiteData@ValueIDs[,to.delete], "And finally remove several value by boolean")

When more than one spatial point and times are given, then use the usually selection without '@ValueIDs'. deleteDataValues(testSiteData[,20:24], "Remove several values.")

Finally RObsDat can show older data-versions.

```
> getDataVersions()

  VersionID          ValidUntil                        VersionComment
1         1 2013-10-23 09:35:34                       Initial Version
2         2 2013-10-23 09:35:34             Correction of wrong value
3         3 2013-10-23 09:35:35                    Changing more data
4         4 2013-10-23 09:35:36 Ups, I used 90 instead of 190 by mistake
5         5                <NA>              And finally remove a value

> versionQuery1 <- getDataValues(Site=1, VersionID=1)
> #stplot(versionQuery1, mode="ts")
>
> versionQuery2 <- getDataValues(Site=1, VersionID=2)
> #stplot(versionQuery2, mode="ts")
```

6 Conclusion

Ideally RObsDat will be a valuable tool for you to manage your data. It is currently still under development. You can contribute by testing the package and reporting what works and what doesn't, by improving the documentation, by making some funding available for further development...

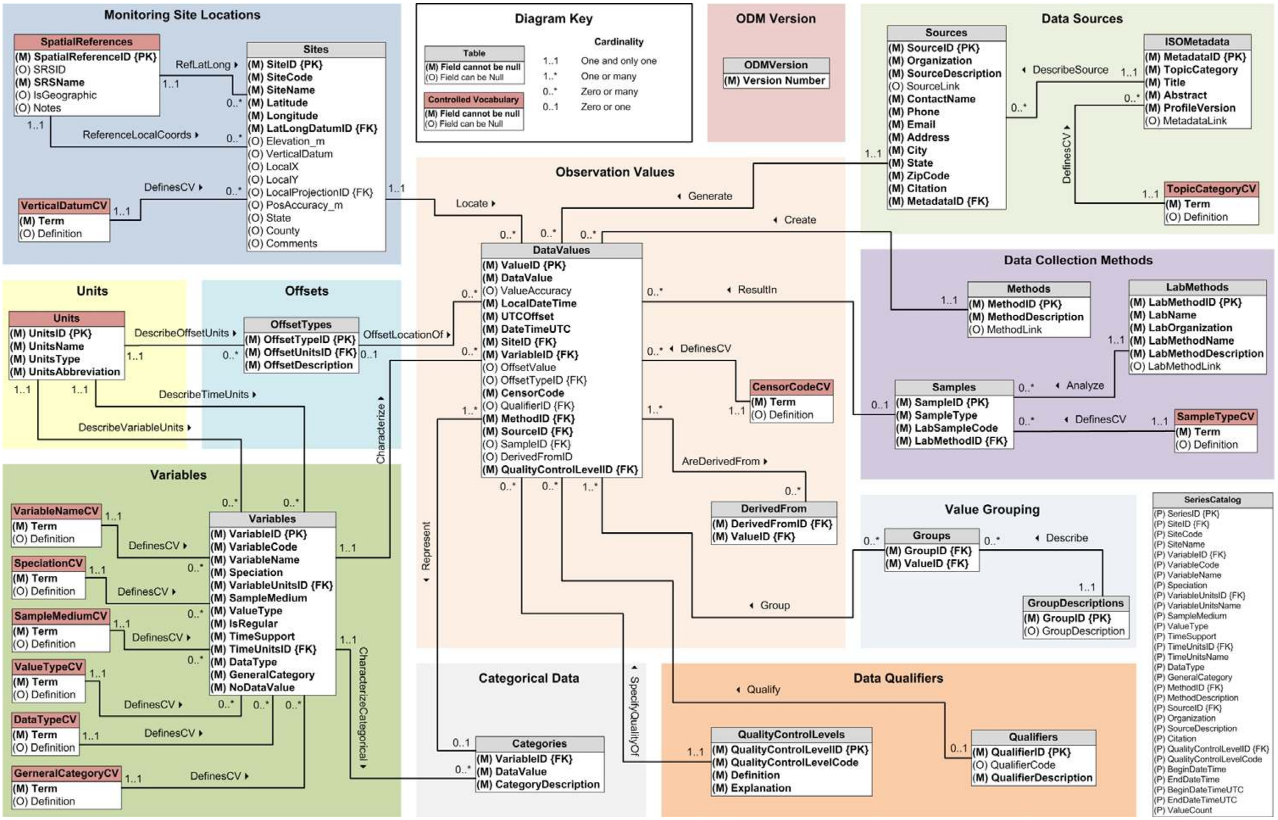

Figure 1: Observations Data Model as specified in Tarboton et al. [2008]. Image source: Tarboton et al. [2008]

A The full data model

References

Anna Gold. Cyberinfrastructure, Data, and Libraries, Part 1: A Cyberinfrastructure Primer for Librarians. *D-Lib Magazine*, 13(9/10):september20september–gold–pt1, 2007. doi: 10.1045/september20september-gold-pt1. URL `http://www.dlib.org/dlib/september07/gold/09gold-pt1.html`.

Edzer J. Pebesma. spacetime: Spatio-Temporal Data in R. *Journal of Statistical Software*, 51(7):1–26, 2012.

David G Tarboton, Jeffery S. Horsburgh, and David R Maidment. CUAHSI Community Observations Data Model (ODM) Version 1.1 Design Specifications, 2008.