aRT: R-TerraLib API

Pedro R. de Andrade Neto
Marcos A. Carrero
Paulo J. Ribeiro Jr

October 10, 2006

Contents

1 Introduction

R is a language and environment for statistical computing and graphics and is freely distributed under the terms of the GNU General Public License [**?**]. It is similar to the S language as originally developed at AT&T Bell Laboratories, although having important differences in the design.

R provides a wide variety of statistical and graphical techniques, is highly extensible having interface with procedures written in C/C++ or FORTRAN. A web site with further information can be found at `http://www.r-project.org`.

TerraLib is a Geographic Information System (GIS) library written in C++, developed at the *Instituto Nacional de Pesquisas Espaciais, Brasil* (INPE), available from the Internet as an open source project, allowing for a collaborative environment for the development of multiple and flexible GIS tools [**?**]. TerraLib defines geographical and temporal data models and provides support for this model over a range of Data-Base Management Systems (DBMS). A web page with further information on TerraLib is available at `http://www.terralib.org`.

An example of application that use TerraLib library of classes is TerraView. This is a Geographical Application tool, with spatial analysis capabilities, and is also licensed as free software under the GNU General Public License. It can be downloaded together with TerraLib.

aRT (API R-TerraLib) is an R package that provides the integration between the softwares R and TerraLib. The idea is to have a package that uses the

statistical analysis provided by R and the geographical data model and database support by TerraLib. A web site with further information can be found at `http://leg.est.ufpr.br/aRT`

The main motivation for the package development is to facilitate the exchanging of information between the spatial packages in R (see `http://sal.agecon.uiuc.edu/csiss/Rgeo/`) and the DBMS using the TerraLib ability to manage and perform some spatial operations on the database. For instance, data can be easily moved between R and TerraLib, and routines in TerraLib can be used to proccess data, making functionalities in TerraLib available to the R packages. This way a data analyst could, for instance, import the data to R, perform some analysis using a spatial package such as `spdep`, `splancs`, `gstat`, `geoR`, among others, and return the results to the database. The results persisted in the database could them be accessed by a GIS software such as TerraView.

aRT is being developed under a `GNU/Linux-Debian` platform and although source code is distributed, there is no guarantee it will work in other one. There are tentatives to mantain a compiled Windows version in the `aRT` web age. The instructions about how to compile and install `aRT` are available at `leg.est.ufpr.br/aRT`.

2   Getting started

After installing `aRT` and starting an R session, load the package with the command `library()`. If the package is loaded successfully a message `TRUE` will be displayed.

```
> library(aRT)
```

```
---------------------------------------------
R-TERRALIB API
http://www.leg.ufpr.br/aRT
TerraLib version 3.2.0 is now loaded
aRT version 1.5-1 (2008-04-09) is now loaded
---------------------------------------------
```

aRT has four main classes to manipulate TerraLib data/functions: `aRTconn`, `aRTdb`, `aRTlayer` and `aRTtheme`. The next subsections explain each class in details. As this is an introductory vignette, we will enable the `aRT` functions message dump, calling `aRTsilent`:

```
> aRTsilent(FALSE)
```

```
[1] FALSE
```

2.1   aRTconn class

Once the package is loaded, we need a DBMS connection. It is encapsulated in an onbect of class **aRTconn**. The constructor of **aRTconn** gets as arguments `user`,

password, `host`, `dbms` and `port`, and their default values are USER variable, empty string, empty string again, the first DBMS found by the configure, and the default port for that DBMS, respectivaly. For example:

```
> con <- openConn(name = "root")

Trying to connect ... yes

> con

Object of class aRTconn

DBMS: "MySQL"
User: "root"
Password: "gpttp"
Port: 3306
Host: ""
```

After creating `con`, the variables it contains cannot be changed. If you need to set them, the only way is to create the object again. It happens because aRT uses external pointers to store the objects, but we will not explain how it works here.

One **aRTconn** object stores a *virtual* connection, i.e., all time that a database access is required, it connects, does something, and then disconnects. The objective of this class is to provide some database administration tasks, and open *real* connections. For example, if it is the first time you are using aRT, perhaps you will need to give permissions to some users. To do so, use `addPermission()`:

```
> addPermission(con, "pedro")

Adding permissions to user 'pedro' ... yes
```

Warning: this function gives ALL permissions in ALL databases to a user. If you want to do something different, you will need to run MySQL (or another DBMS) for yourself, and use the `GRANT` command.

With an **aRTconn** object, you can also see the databases available and remove them. The next example shows the databases and tries to remove a database called `bodmin` if it exists:

```
> showDbs(con)

 [1] "information_schema" "auckland"           "bodmin"
 [4] "ca20"               "catarina"           "dynatt"
 [7] "image"              "meso"               "mysql"
[10] "parana"             "pol3"               "recife"
[13] "saudavel"           "sp"                 "tabletest"

> if (any(showDbs(con) == "bodmin")) deleteDb(con, "bodmin", force = TRUE)

Checking for database 'bodmin' ... yes
Deleting database 'bodmin' ... yes
```

2.2 aRTdb class

To create a new database, or to access one, there is the **aRTdb** class. One object from this class stores a *real* database connection, and we need an **aRTconn** object to create it:

```
> db <- createDb(con, db = "bodmin")

Creating database 'bodmin' ... yes
Creating conceptual model of database 'bodmin' ... yes
yes
Loading layer set of database 'bodmin' ... yes
Loading 'root' view set of database 'bodmin' ... yes

> db

Object of class aRTdb

Database: "bodmin"
Layers: (none)
Themes: (none)
External tables: (none)
```

This constructor fails if the database already exists. Once this object is created, it depends no more on the **con** object.

A **aRTdb** object contains *all* TerraLib objects in memory needed by **aRT**. This means that all objects opened from this one depends on it, even after they are created in R. If this object is removed from R, all his "childrens" become invalid objects when R's garbage collector remove this object from memory.

2.3 aRTlayer class

To work with data in aRT, we need to manipulate *layers*. A layer can store any geometry of one kind (points, lines, polygons, raster and cells), and attributes. Layers are TerraLib abstractions that use tables of data and tables of control in one database. So they can be created from **aRTdb** objects.

```
> layer.points <- createLayer(db, "points")

Checking for layer 'points' ... no
Building projection to layer 'points' ... yes
Creating layer 'points' ... yes

> layer.points

Object of class aRTlayer

Layer: "points"
```

```
Database: "bodmin"
Layer is empty
Projection Name: "NoProjection"
Projection Datum: "Spherical"
Tables: (none)
```

To insert data in the layer, we will use the bodmin dataset, available within
splancs package.

```
> require(splancs)

Spatial Point Pattern Analysis Code in S-Plus

 Version 2 - Spatial and Space-Time analysis

> data(bodmin)
> names(bodmin)

[1] "x"    "y"    "area" "poly"
```

Before insert into the database, we must convert the data to aRT format,
following sp classes. The next commands converts it to a SpatialPoints-
DataFrame and inserts it into the database:

```
> SPoints = SpatialPointsDataFrame(cbind(bodmin$x, bodmin$y), data.frame(ID = paste(1:le
> addPoints(layer.points, SPoints)

Converting points to TerraLib format ... yes
Adding 35 points to layer 'points' ... yes
Reloading tables of layer 'points' ... yes

> t = createTable(layer.points, "tpoints")

Creating table tpoints ... Creating static table 'tpoints' on layer 'points' ... yes
Creating link ids ... yes
yes

> layer.points

Object of class aRTlayer

Layer: "points"
Database: "bodmin"
Number of points: 35
Projection Name: "NoProjection"
Projection Datum: "Spherical"
Tables:
    "tpoints": static
```

To insert the evolving polygon, we will create another layer:

```
> p = Polygon(bodmin$poly)
> P = Polygons(list(p), ID = "1")
> SP = SpatialPolygons(list(P))
> layer.pol <- createLayer(db, l = "polygons")

Checking for layer 'polygons' ... no
Building projection to layer 'polygons' ... yes
Creating layer 'polygons' ... yes

> addPolygons(layer.pol, SP)

Converting polygons to TerraLib format ... yes
Adding 1 polygons to layer 'polygons' ... yes
Reloading tables of layer 'polygons' ... yes

> t = createTable(layer.pol, "tpol")

Creating table tpol ... Creating static table 'tpol' on layer 'polygons' ... yes
Creating link ids ... yes
yes
```

Finally we will do a kernel analysis, and insert the raster data into the database, in another layer:

```
> raster <- kernel2d(as.points(bodmin), bodmin$poly, h0 = 2, nx = 100,
+     ny = 200)

Xrange is  -5.2 9.5
Yrange is  -11.5 8.3
Doing quartic kernel

> layer.raster <- createLayer(db, l = "raster")

Checking for layer 'raster' ... no
Building projection to layer 'raster' ... yes
Creating layer 'raster' ... yes

> addRaster(layer.raster, raster)

Initializing the raster ... yes
Adding raster data to layer 'raster' ... yes
Reloading tables of layer 'raster' ... yes
```

Finally, there are three layers in the database, and they can be seen in the next code:
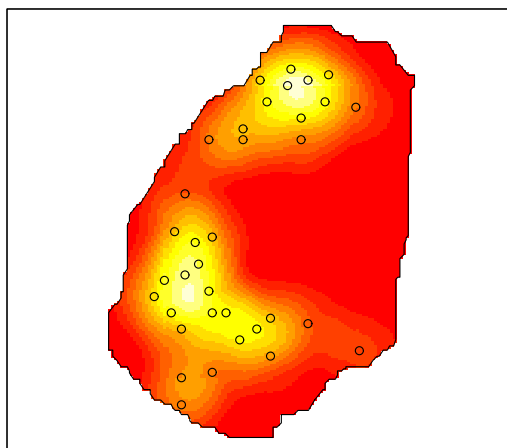
```
> showLayers(db)
```

Figure 1: Data from the three layers

```
[1] "points"   "polygons" "raster"

> db

Object of class aRTdb

Database: "bodmin"
Layers:
    "points"
    "polygons"
    "raster"
Themes: (none)
External tables: (none)
```

To get the layer's geometry call **getGeometry**, and then you can plot it. But if you don't need the data the layer can be plotted directly:

```
> plot(layer.raster)
> plot(layer.points, add = TRUE)
> pols = getPolygons(layer.pol)
> plot(pols, add = TRUE)
```

2.4 aRTtheme class

The last class implemented in `aRT` is `aRTtheme`. Themes can be visualized in TerraView software, and they can select data and join tables. For now, we will only create themes of points and polygons, and put them in the view `view`:

```
> theme.points <- createTheme(layer.points, t = "points", view = "view")

Checking for theme 'points' in layer 'bodmin' ... no
Checking for view 'view' in database 'bodmin' ... no
Creating view 'view' ... yes
Inserting view 'view' in database 'bodmin' ... yes
Creating theme 'points' on layer 'points' ... yes
Checking tables of theme 'points' ... yes
Saving theme 'points' ... yes
Building collection of theme 'points' ... yes
Generating positions of theme 'points' ... yes

> setVisual(theme.points, visualPoints(size = 5))
> theme.pol <- createTheme(layer.pol, t = "polygons", view = "view")

Checking for theme 'polygons' in layer 'bodmin' ... no
Checking for view 'view' in database 'bodmin' ... yes
Creating theme 'polygons' on layer 'polygons' ... yes
Checking tables of theme 'polygons' ... yes
Saving theme 'polygons' ... yes
Building collection of theme 'polygons' ... yes
Generating positions of theme 'polygons' ... yes

> setVisual(theme.pol, visualPolygons())
```

There is an argument that can be used in raster themes: the colors configuration. It can be used as in the next example.

```
> theme.raster <- createTheme(layer.raster, t = "raster", v = "view")

Checking for theme 'raster' in layer 'bodmin' ... no
Checking for view 'view' in database 'bodmin' ... yes
Creating theme 'raster' on layer 'raster' ... yes
Checking tables of theme 'raster' ... yes
Saving theme 'raster' ... yes

> setVisual(theme.raster, visualRaster(col = terrain.colors(20)),
+     mode = "raster")
```