aRT: R-TerraLib API

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

December 20, 2005

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 which was developed at AT&T Bell Laboratories, but they have important differences in the designs.

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 by *Instituto Nacional de Pesquisas Espaciais* (INPE), available from the Internet as open source, allowing a collaborative environment and its use for the development of multiple GIS tools [**?**]. It defines a geographical data model and provides support for this model over a range of different Data-Base Management Systems (DBMS). A web site with further information can be found at `http://www.terralib.org`.

An example of application that use TerraLib class library is TerraView. It 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 a package that provides the integration between the softwares R and TerraLib. aRT is still a prototype with some basic operations which prove that the integration is possible. 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://www.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 TerraLib ability to manage and perform some spatial operations on the database. For instance, data can be easily moved between R and TerraLib. 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. Those results could them be accessed by a GIS software such as TerraView.

Section 2 lists the aRT requirements and dependencies. To start with the development we have defined six basic operations which are described in Section 3 with examples of the capabilities of aRT.

2 Requirements and dependencies

aRT is being developed under a GNU/Linux-Debian platform and do not have guarantees to work in other one. This prototype still doesn't have (yet...) an *autoconfigure*, so the configuration must be done manually. The following softwares/libraries are necessary:

- MySQL DBMS version 14.7 or upper and libraries:
  http://www.mysql.com
- Qt library version 3.3.3 or upper (Qt is a multiplataform toolkit in C++ for the development of graphic interfaces, implemented by Trolltech:)
  http://www.trolltech.com
  Qt library is used to TerraLib compilation.
- TerraLib 3.0.2 or upper:
  http://www.terralib.org
  TerraLib library is generated through the compilation.
- R language:
  http://cran.r-project.org
- GNU gcc/g++ compiler, MAKE
  http://www.gnu.org
- aRT package:
  http://www.est.ufpr.br/aRT

There is a script to download and install MySQL/Qt/TerraLib under Debian/Linux along with the package.

Once we installed MySQL, Qt, TerraLib and R in the directory /usr/local, the following environment variables should be placed, i.e, in the .bash_profile or .bashrc file of the user's login directory. Change the directories according with the installations' pathes.

```
# Default directories (TerraLib, libmysqlclient.a and libR.so),
# used to __Make__ aRT:

TERRALIBDIR=/usr/local/terralib
LIBMYSQLCLIENTDIR=/usr/lib
LIBRDIR=/usr/local/lib/R/lib

# TerraLib Shared libraries, used to __execute__ aRT:

LD_LIBRARY_PATH=$TERRALIBDIR/terralibx/terralib:\
$TERRALIBDIR/terralibx/tiff:\
$TERRALIBDIR/terralibx/shapelib:\
$TERRALIBDIR/terralibx/stat

export TERRALIBDIR MYSQLDIR LD_LIBRARY_PATH LIBRDIR
```

3 Getting started

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

```
> library(aRT)

Loading required package: sp

------------------------------------------------
R-TERRALIB API
aRT version 0.4-15 (2005-12-20) is now loaded
------------------------------------------------
```

aRT has four classes to manipulate TerraLib data/functions: aRTconn, aRTdb, aRTlayer and aRTtheme. The next subsections explain each class in details.

3.1 aRTconn class

Once the package is loaded, we need a DBMS connection. It can be done creating an aRTconn object. The constructor of aRTconn class gets the arguments user, password, host and port, and their default values are USER variable, empty string, "localhost" and 3306, respectivaly. For example:

```
> con <- openConn(user = "root", host = "localhost", pass = "")

Trying to connect ... yes

> print(con)

Object of class aRTconn

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

After the object `con` is created, the variables it contains cannot be changed. If you need to set them, the only way is to create another object. This occours because data is stored in a external pointer, but we will not explain these things here.

One `aRTconn` object stores a *virtual* connection, i.e., all time that a database access is required, it connects, do something, and then disconnect. The objective of this class is to do some database administration functions, and open *real* connections. For example, if it is the first time you are running `aRT`, maybe you need to add permissions to some users. To do this, use `addUser()`:

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

Adding user 'pedro' ... yes
```

Warning: this function gives ALL permissions to a user. If you want to do something different, you need to run mysql 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 the database parana if it exists:

```
> showDbs(con)

 [1] "Parana"        "auck"          "bodmin"        "ca20"
 [5] "catarina"      "citrusTeste10" "dbsjc"         "dummy"
 [9] "dynatt"        "event"         "intro"         "leg"
[13] "meso"          "mysql"         "parana"        "pol3"
[17] "pr"            "recife"        "saudavel"      "sp"
[21] "t4"            "tabletest"     "test"          "testeparana"
[25] "tyr"

> if (any(showDbs(con) == "bodmin")) deleteDb(con, "bodmin", f = T)

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

3.2 aRTdb class

To create a new database, or to access one, there is the `aRTdb` class. Objects from this class stores a *real* database connection, and needs an `aRTconn` object to be created:

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

Connecting with database 'bodmin' ... no
Creating database 'bodmin' ... yes
Creating conceptual model of database 'bodmin' ... yes
Loading layer set of database 'bodmin' ... yes
Loading view set of database 'bodmin' ... yes

> print(db)

Object of class aRTdb

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

This object has 0 children
```

This constructor tries to load a database with name `parana`. If it does not exists (the true, once we removed it), it checks for `create`, trying to create a new one. Once this object is created, it depends no more of the `con` object.

`aRTdb` objects contains *all* TerraLib objects in memory needed by `aRT`. This means that all objects opened from it depends on it, even after they are created in R. The last line of `print` shows the number of children this object has. If this object is removed from R, all childrens becomes invalid objects when R's garbage collector remove this object from memory.

3.3 aRTlayer class

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

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

Building projection to layer 'points' ... yes
Creating layer 'points' ... yes

> print(layer.points)
```

```
Object of class aRTlayer

Layer: "points"
Database: "bodmin"
Number of polygons: 0
Number of lines: 0
Number of points: 0
Layer does not have raster data
Projection Name: "NoProjection"
Projection Datum: "Spherical"
Projection Longitude: 0
Projection Latitude: 0
Tables: (none)
```

There is an argument `proj` in the constructor that says which projection the layer data is. The defaul value is `plan`, meaning that the data can be drawn as it is. The other option (until now) is `gepgraphic`, meaning that the data is in degrees. Then we need to convert the data before plot it. ((Referência?? Simone??))

To insert data in the layer, we will use the bodmin dataset, part of `splancs` package.

```
> require(splancs)

Loading required package: splancs

Spatial Point Pattern Analysis Code in S-Plus

 Version 2 - Spatial and Space-Time analysis

Attaching package: 'splancs'


        The following object(s) are masked from package:sp :

         bbox

[1] TRUE

> data(bodmin)
> names(bodmin)

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

Before insert into the database, we must convert the data to `aRT` format. `aRT` has some functions to convert data from other spatial packages (`splancs` and `geoR`, actually). This functions have the format `<pkg>2aRT<datatype>`,

where pkg can be sp or gr, and datatype can be one of points, polygons or attributes. As example, the next code converts bodmin data from splancs to aRT, and inserts it into the database[1]:

```
> SPoints = SpatialPointsDataFrame(cbind(bodmin$x, bodmin$y), data.frame(ID = paste(1:ler
> 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", gen = T)

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

> print(layer.points)

Object of class aRTlayer

Layer: "points"
Database: "bodmin"
Number of polygons: 0
Number of lines: 0
Number of points: 35
Layer does not have raster data
Projection Name: "NoProjection"
Projection Datum: "Spherical"
Projection Longitude: 0
Projection Latitude: 0
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")

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
```

[1] You can disable the aRT functions message dump calling aRTsilent(TRUE)

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

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

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
```

Finnaly, there are three layers created, children of `db`, as can be seen in the next code:

```
> showLayers(db)

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

> print(db)

Object of class aRTdb

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

This object has 5 children
```

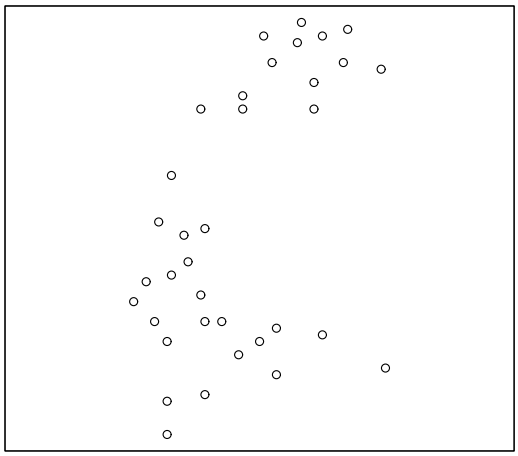

Figure 1: Point set in `layer.points`

All `add` functions recieve an argument `close = TRUE`, telling if it is the last time the data will be inserted in the layer. Once the layer is closed, no geometry can be added to it[2]. You can implicitly close the layer and create the table calling `createEmptyTable()`. After close the layer, attributes can be inserted.

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.points)
```

```
> pols = getPolygons(layer.pol)
> plot(pols)
```

```
> plot(layer.raster)
```

You can plot different layers, using `add = TRUE`.

[2]until now, there is no protection to one who tries it, but no attributes of the geometry inserted after close the layer can be retrieved.

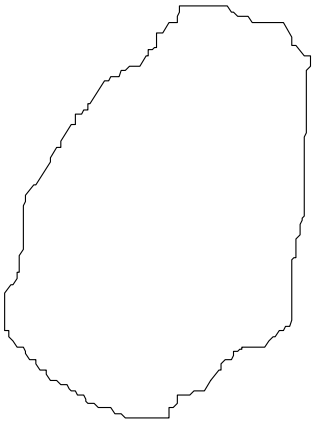

Figure 2: Evolving polygon in `layer.pol`

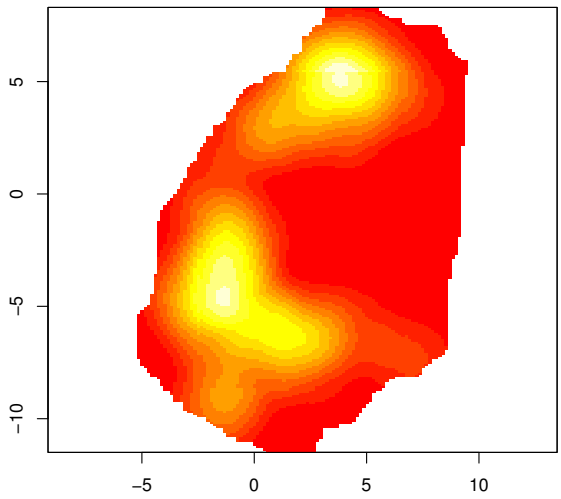

Figure 3: Raster data from `layer.raster`

3.4 aRTtheme class (TerraView users only)

The last class implemented in aRT is aRTtheme. Themes can be visualized in TerraView software, and are (until now) useless for non-TerraView users.

Now we will create themes of points and polygons, and put them in the view view:

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

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

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

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

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

3.5 Spatial and attributes data format

3.6 Removing

There are two kinds of removing aRT objects: from memory and from database. aRTdb objects stores all the memory of aRT objects. aRTlayer only have pointers

to their `aRTdb`. To remove data from memory is just call `rm`, and (for `aRTdb`'s), if you want to free the memory call `gc` explicitaly. Note that, once an `aRTlayer` object needs an aRTdb, if the garbage collector removes one `aRTdb` object, all the `aRTlayer`s opened from it will become invalid.