

GRASS vector data management and operation enhancement by using RDBMS PostgreSQL 2–d vectors.

Alexander Shevlakov*

* Free Software consulting Motivation , Pushkovikh St., 9, of.106 , 142092 Troitsk, Moscow, Russia, tel. +79021739952, fax +70957773992, e–mail Alex.Shevlakov@motivation.ru

1 Introduction

This paper describes a mechanism for integrating GIS GRASS 5.0 and RDBMS PostgreSQL in one tool for vector processing, GRASS vector maps management, various spatial geometry operations on two dimensional vector maps. It is for well known platforms such as ARC/INFO and Oracle that we mostly find existing software applications that involve GIS–RDBMS linkage or address ourselves to in assessments of GIS effectiveness.

However, one of the most advantageous features of free RDBMS software PostgreSQL is implementation of geometry types, rules, operators and functions [2], making it possible to use database for keeping and processing data in the same way as inside a GIS, due to the fact that basic geometry types are common for both systems. One part of the data processing may be done by database, and the other by an interfaced GIS, including visualization..

Considering proprietary software high costs, both in GIS and RDBMS, it is clear that the potential for free GIS and database management software offer to end users is growing. Various examples of GRASS modules, some of them new in GRASS 5.0, interacting with PostgreSQL are shown below, some of them using the geometry functions that are not part of GRASS 5.0 distribution neither of PostgreSQL and must be thought of as an interface between them. As three–dimensional vector support appears in GRASS today and more complex tools are needed to process data during computations [3], the usage of PostgreSQL built–in geometry queries and contribution libraries plays an increasingly important role.

2 Study area

1. The area of over 100,000 hectares situated in the coastal part of the Sikhote–Alin State Biosphere Nature reserve (Russian Federation) was selected for this study as up to the moment, there had been large GRASS datasets for this area, including vector maps digitized from 1:25000 topography maps, and in PostgreSQL, relations holding entries for each of 6,893 polygons of the land vector map (Tab. 3) and for 484 streams (Tab. 4).

The real world coordinates of the maps are plane x,y Gauss–Krueger coordinates projected from the Krassovsky ellipsoid, Pulkovo datum (S–42); though maps were digitized at a larger scale, they were georeferenced from 1:100,000 scale topography maps (Fig. 1).

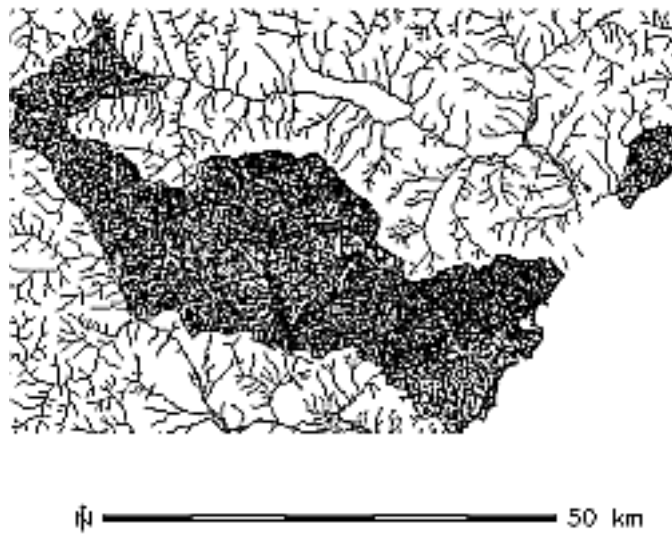


Figure 1: The study area map with 6893 polygons and streams map with 1873 lines.

The study area is covered with forest (97 percent), inside the area elevations range from zero (coast of the Sea of Japan) to 1,500 m above sea level. The plots of the land map were delineated with respect to the vegetation, slope and aspect of the given territory by the forest inventory in 1979 with use of the aerial photos. The plots sizes range from 0.1 hectares to 238 hectares, 14 hectares average (Fig.2).

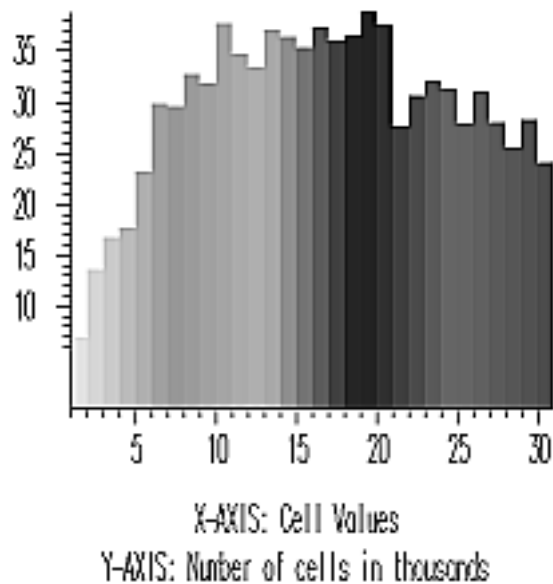


Figure 2: Histogram of plot size values (in hectares).

The hydrographic network (Fig.3) consists of streams running from west to east, from slopes of the Sikhote–Alin range into the Sea of Japan. The mean density of the network is 0.7 km per sq. km. It is characteristic of these streams to have V–formed valleys, narrow (to 50–60

m) and rapid (1–3 m/sec) course, sometimes waterfalls. The major rivers (Dzhiguitovka, Serebryanka) near the sea flow through large estuary valleys [1].



Figure 3: The hydrography network.

3 Description of Methods

I. First of all, we create two new tables in Postgres for polygons (land areas) with their boundary coordinates, topology descriptors and categories ('info_kuruma_bnd'), and lines (streams) from inside the study area with line coordinates and categories ('riv_names_arc'), by using the GRASS–PostgreSQL interface module v.to.pg:

```
g.region map=kuruma_id
v.to.pg -t kuruma_id type=area tab=info_kuruma key=rec_id
v.to.pg -t rivers type=line tab=riv_names key=riv_id
```

The new tables have 'key' (= category) field named the same as category field in their reference tables, i.e., 'rec_id', in table 'info_kuruma' which contains other attribute information for land areas (such as vegetation type index field 'type_id'), and 'riv_id' in 'riv_names', for streams. The new tables can be linked to other tables by this category field.

Table 1. The fields of relation info_kuruma_bnd.

<i>Attribute</i>	<i>Type</i>	<i>Modifier</i>
<i>rec_id</i>	<i>integer</i>	<i>/</i>
<i>num</i>	<i>smallint</i>	<i>/</i>
<i>ex</i>	<i>boolean</i>	<i>/</i>
<i>boundary</i>	<i>polygon</i>	<i>/</i>

Table 2. The fields of relation riv_names_arc.

<i>Attribute</i>	<i>Type</i>	<i>Modifier</i>
<i>riv_id</i>	<i>integer</i>	<i>/</i>
<i>num</i>	<i>smallint</i>	<i>/</i>
<i>segment</i>	<i>path</i>	<i>/</i>

Although the names of the reference tables were given as parameter 'tab' before import, this had no effect during the insertion of lines, as it otherwise would unless the flag '-t' (total) was used. The table 'riv_names' had fewer entries than the number of segments in map 'rivers', because small tributories had not been named, and therefore were not present in the 'riv_names' table. Those streams were indexed '399999' in the vector map 'rivers'. If the tables had been referenced during import (i.e., without '-t' flag), only those streams that had corresponding index 'riv_id' in table 'riv_names' would have been exported into new Postgres table.

Table 3. The fields of relation info_kuruma

<i>Attribute</i>	<i>Type</i>	<i>Modifier</i>
<i>area</i>	<i>real</i>	<i>/</i>
<i>perim</i>	<i>real</i>	<i>/</i>
<i>rec_id</i>	<i>integer</i>	<i>/</i>
<i>usl_proiz</i>	<i>smallint</i>	<i>/</i>
<i>main</i>	<i>character(4)</i>	<i>/</i>
<i>hight</i>	<i>smallint</i>	<i>/</i>
<i>diam</i>	<i>smallint</i>	<i>/</i>

Table 4. The fields of relation riv_names

<i>Attribute</i>	<i>Type</i>	<i>Modifier</i>
<i>riv_id</i>	<i>integer</i>	<i>/</i>
<i>riv_names</i>	<i>text</i>	<i>/</i>

There should be more polygons in the new table than in the reference table, because all extra polygons are "holes", and therefore have index 'f' in the boolean field 'ex':

```
sixote=> select count(1) from info_kuruma;
count
-----
  6893
(1 row)
```

```
sixote=> select count(1) from info_kuruma_bnd;
count
-----
  7336
(1 row)
```

```
sixote=> select count(1) from info_kuruma_bnd where ex = 'f';
count
-----
   443
(1 row)
```

Also, there are more segments in table 'riv_names_arc' than indexed in the reference table:

```
sixote=> select count(1) from riv_names;
count
-----
   484
(1 row)
sixote=> select count(1) from riv_names_arc;
count
-----
  1873
(1 row)
```

where all extra lines are those with index '399999' (i.e., nameless).

II. Perform example PostgreSQL spatial queries on area relation 'info_kuruma_bnd':

A.

Select only polygons with BBox within a 5-km-wide ring with center in point(x=600000, y=4984000) (Fig.4):

```
select rec_id from info_kuruma_bnd where ((boundary::box <-> point ('(600000,
4984000)')) < 10000 and (boundary::box <-> point ('(600000, 4984000)')) > 5000);
```

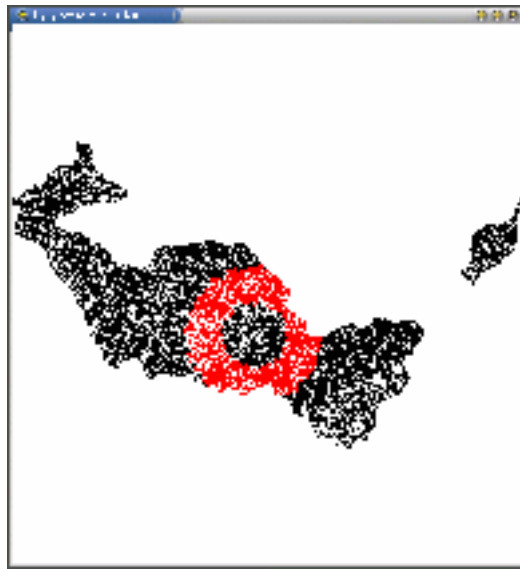


Figure 4: Selected polygons fill the "ring".

B.

Select only polygons whose BBox is within three 10-km-wide vertical bands (Fig.5):

```
select rec_id from info_kuruma_bnd where (boundary::box << point ('(620000, 4984000)')
and boundary::box >> point ('(610000, 4984000)')) or (boundary::box << point ('(600000,
4984000)') and boundary::box >> point ('(590000, 4984000)')) or (boundary::box <<
point ('(580000, 4984000)') and boundary::box >> point ('(570000, 4984000)'));
```

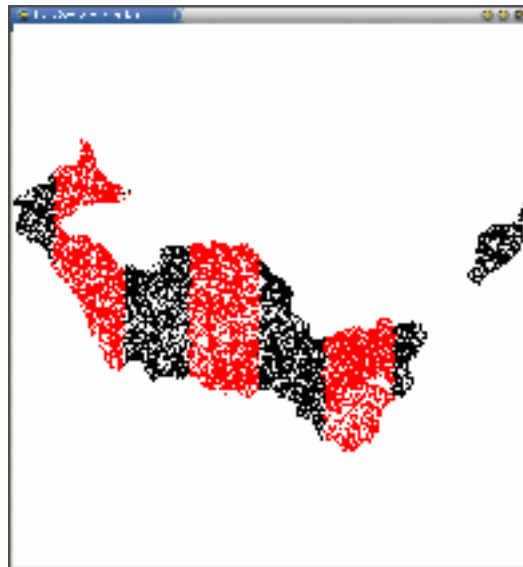


Figure 5: Selected polygons fill the "stripes".

C.

Select only polygons with BBox within three 10–km–wide stripes with 30 degrees rotation clockwise (Fig.6):

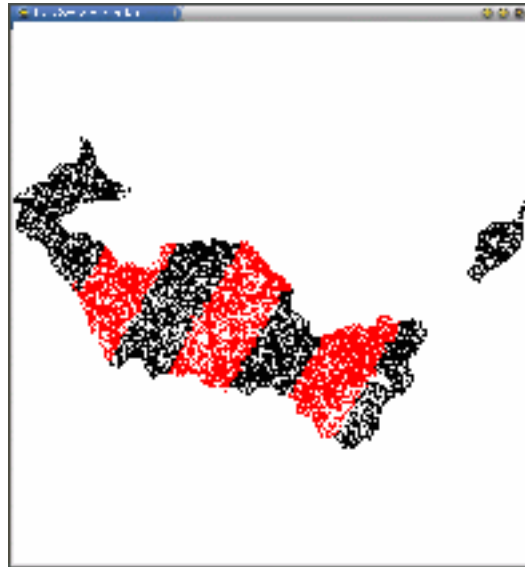


Figure 6: Selected polygons fill the "skewed stripes".

```
select rec_id from info_kuruma_bnd where (boundary::box * point ('(.866,.5)') << point ('(620000, 4984000)') * point ('(.866,.5)') and boundary::box * point ('(.866,.5)') >> point ('(610000, 4984000)') * point ('(.866,.5)')) or (boundary::box * point ('(.866,.5)') << point ('(600000, 4984000)') * point ('(.866,.5)') and boundary::box * point ('(.866,.5)') >> point ('(590000, 4984000)') * point ('(.866,.5)')) or (boundary::box * point ('(.866,.5)') << point ('(580000, 4984000)') * point ('(.866,.5)') and boundary::box * point ('(.866,.5)') >> point ('(570000, 4984000)') * point ('(.866,.5)'));
```

D.

Selects only polygons whose BBox is within three 30–degree–wide sector with 30 degrees rotation clockwise from point (x=590000, y=4980000) (Fig.7):

```
select rec_id from info_kuruma_bnd where (boundary::box * point ('(.866,.5)')>> point ('(590000, 4980000)') * point ('(.866,.5)') and boundary::box * point ('(.5,.866)')<< point ('(590000, 4980000)') * point ('(.5,.866)'))
```

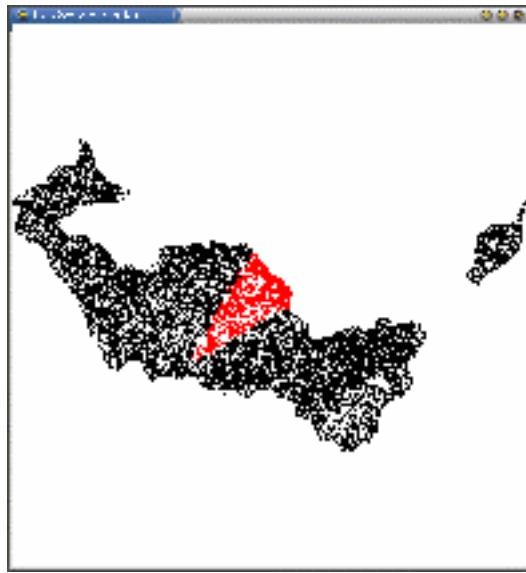


Figure 7: Selected polygons fill the "sector".

The results of the queries shown above can be retrieved in form of new GRASS vector maps, if necessary. This is done by module `d.vect.pg` with a new option `'-e'` – extract:

```
d.vect.pg -e -s sql=file.sql map=kuruma_id
```

III. Build buffer zones around selected lines:

The computation algorithm used to construct the buffer vector line is based on two adjacent segments of main polyline, finding the tangent of angle between them and determining the oblique side of the angle. Then intersections of upper and lower buffer line segments were computed to produce vertices of the buffer. In the example given below, the streams map was digitized at 1:25,000 scale with average of 217 vertices per line, average segment length of 40 m, while the buffer zone had the radius of 300 m, i.e., about ten times more. This fact is important because the limit of this algorithm being strictly exact depends on buffer radius comparison to the minimal segment length.

First of all, we should select two streams (No.205001, Golubichnaya, and No. 305032, Levyi) and extract them into a new vector map `'two_riv'` (Fig.8):

```
d.vect.pg -e map=rivers tab=riv_names key=riv_id color=blue where='riv_id=205001 or riv_id=305032'
```



Figure 8: Two streams to be extracted from base map.

Then we should create a new Postgres table 'two_riv_arc' for those two streams:

```
v.to.pg -t map=two_riv key=riv_id tab=two_riv type=line
```

Various geometry functions are available in PostgreSQL 'as is', for example, right here we are able to query the streams lengths:

```
sixote=> select riv_id, length(segment) from two_riv_arc;
```

```
riv_id      | length
-----+-----
205001      | 10089.4375378248
305032      | 7366.85013250007
(records: 2)
```

The full list of PostgreSQL functions and operators is available in its distribution documents catalog. Building buffer zones, however, requires additional functions that are supplementary to PostgreSQL distribution and can be loaded as shared library compiled independently; the source code is file called "geo_addfuncs.c" to be compiled, for example, in Linux as follows:

```
gcc -I/postgreSQL-dist/src/include -c -o geo_addfuncs.o geo_addfuncs.c
```

and

```
ld -shared -o geo_addfuncs.so geo_addfuncs.o
```

The result is the shared library 'geo_addfuncs.so', and it can be used to create new functions

within PostgreSQL installation as follows:

```
a) CREATE FUNCTION pt_in_path_buffer(path,point,float8) RETURNS boolean AS
'/home/postgres/geo_addfuncs.so', 'path_buffer_contain_pt' LANGUAGE 'c';
```

The new function 'pt_in_path_buffer' takes three arguments – path, point, and buffer radius, and returns 'yes' or 'no' depending on the given point is within buffer zone of given radius built from parameter 'path'. Let's see how this new function works:

```
sixote=# select riv_id,pt_in_path_buffer(segment,'614680.375,4980083.75',500) from
two_riv_arc;
```

```
riv_id      | pt_in_path_buffer
-----+-----
205001      | t
305032      | f
(records: 2)
```

Obviously, this means the point with given coordinates (x,y): '614680.375,4980083.75' lies within 500 m buffer from the Golubichnaya (#205001) and beyond the zone around the Levyi (#305032).

To build buffer zones, we shall need other functions, namely 'path_to_file', 'return_buffer' and 'path_reduce'. We create them as follows:

```
b) CREATE FUNCTION return_buffer(path,float8) RETURNS path AS
'/home/postgres/geo_addfuncs.so', 'return_path_buffer' LANGUAGE 'c';
```

This new function creates and returns a PostgreSQL 'path' element of the buffer zone edge.

```
c) CREATE FUNCTION path_reduce(path,int2) RETURNS path AS
'/home/postgres/geo_addfuncs.so', 'reduce_path_points' LANGUAGE 'c';
```

This new function takes a 'path' as parameter and generalizes it so many times as the second integer parameter 'i' is. The generalization is done by preserving each i-th point in path discarding others.

```
d) CREATE FUNCTION path_to_file(path) RETURNS boolean AS
'/home/postgres/geo_addfuncs.so', 'write_path_to_file' LANGUAGE 'c';
```

This new function writes the PostgreSQL 'path' type element into a text file with format suitable for import to GRASS with module v.in.arc. It returns true or false depending on select criterion used in a query.

Now we are ready to produce buffer zones in PostgreSQL and import them back to GRASS. This is done by an SQL command:

```
SELECT path_to_file(return_buffer(path_reduce(segment,12),300)) FROM two_riv_arc
WHERE riv_id = 205001;
```

Then in GRASS:

```
v.in.arc type=polygon lines_in='/tmp/boundary.pol' vector_out=a300  
v.support a300
```

We repeat this procedure for the other stream:

```
SELECT path_to_file(return_buffer(path_reduce(segment,12),300)) FROM two_riv_arc  
WHERE riv_id = 305032;
```

```
v.in.arc type=polygon lines_in='/tmp/boundary.pol' vector_out=b300  
v.support b300
```

Finally, we patch two vector maps (v.patch) and remove the inner lines (v.digit) to produce a total 300 m buffer around both streams (Fig. 9).

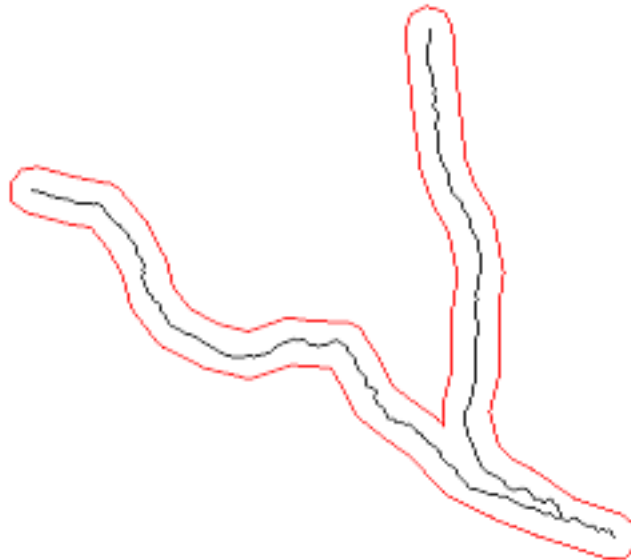


Figure 9: Buffer zones built around two streams.

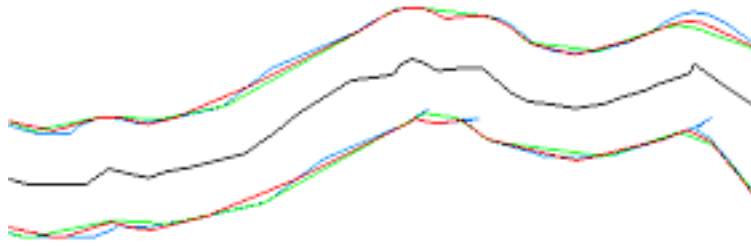


Figure 10: Buffer line smoothness parameter.

The generalization parameter equal to 12 was chosen to obtain smooth contours of the buffers, as it is only possible to compute true buffer contours by using this algorithm if all digitized segments lengths are greater than buffer zone radius. Fig.10 shows how the 'swallow tails' are eliminated by increasing this parameter, and how simultaneously the buffer edge deflects from true position. So if we want building exact buffer zones (i.e., with generalization parameter set to one), this algorithm should not be applied if buffer radius is greater than minimal path segment length.

References

- [1] V.V.Vetrennikov. The geology of the Sikhote–Alin Reserve and the Central Sikhote–Alin ridge. (in Russian language). Vladivostok, 1976, 167 pp.
- [2] PostgreSQL: Introduction and Concepts, by Bruce Momjian , Addison–Wesley, 2000. 462 pp.
- [3] OpenGIS Simple Features Specification for SQL, rev. 1.1 . OpenGIS Project Document 99–049

2

*GRASS vector layer management and operation enhancement by using RDBMS PostgreSQL
2–d vectors .*

3

Alexander Shevlakov