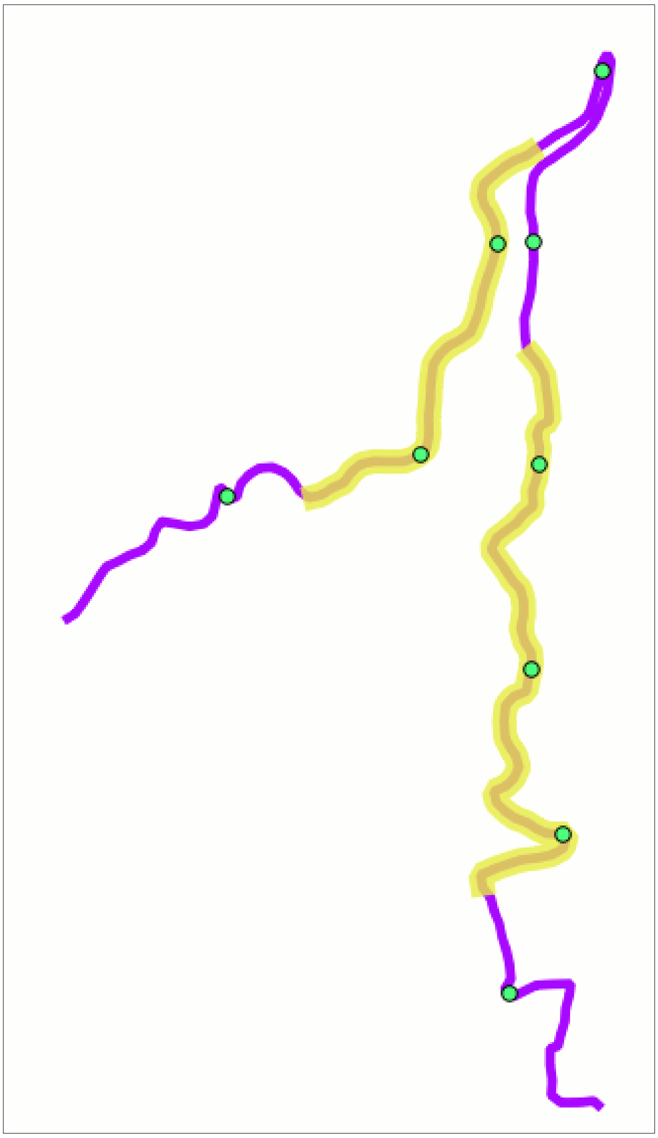
Supporting coolest GEOS v.3.3.0 advanced features
(*and other miscellaneous stuff*)

`ST_Line_Locate_Point()` / `ST_Line_Substring()`



We'll start from a quite complex LINESTRING [*violet line*]: for the sake of simplicity we'll call such LINESTRING as *the_line*.

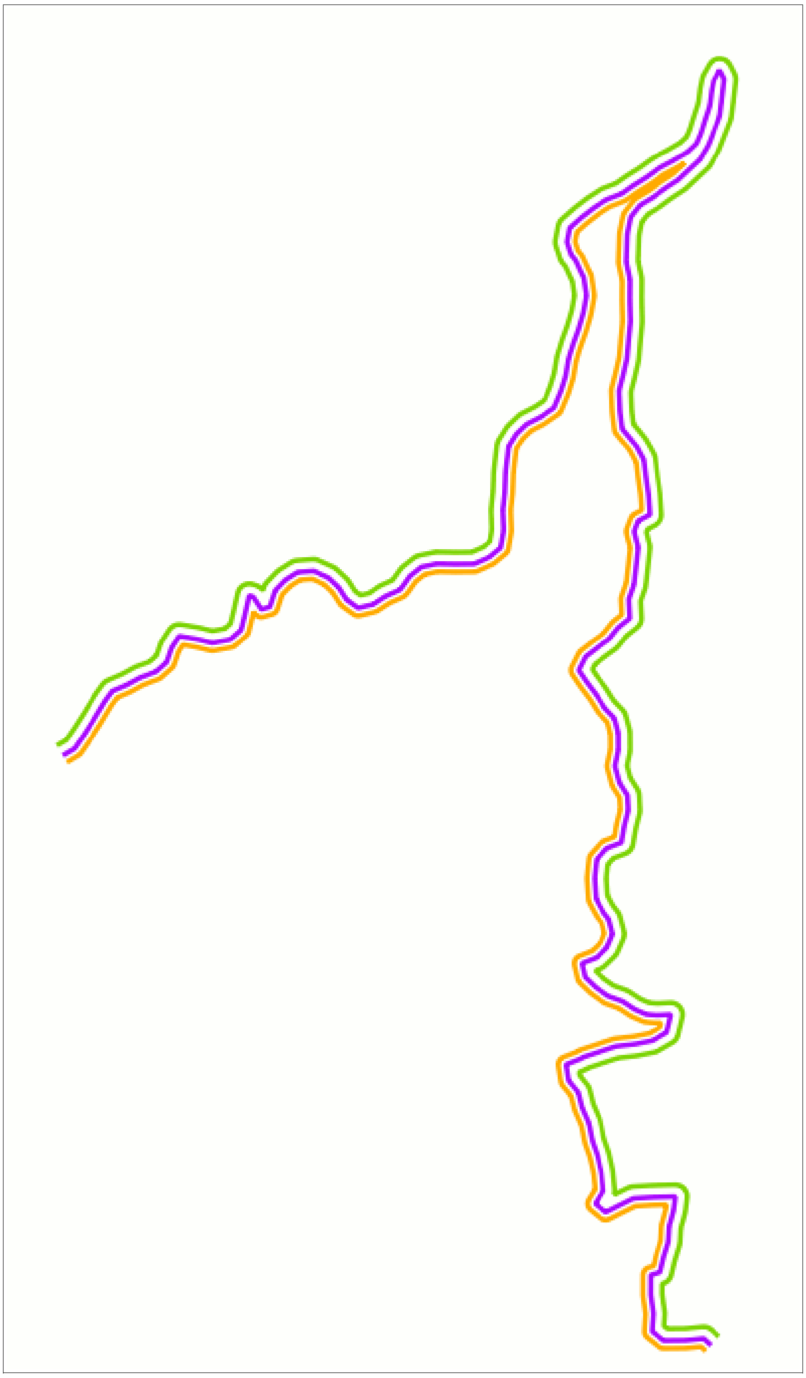
```
SELECT ST_Line_Interpolate_Point(the_line, 0.1);
SELECT ST_Line_Interpolate_Point(the_line, 0.2);
...
SELECT ST_Line_Interpolate_Point(the_line, 0.9);
```

These functions will create a POINT laying on the_line every 10% of the total line length [*green dots*]

```
SELECT ST_Line_Substring(the_geom, 0.15, 0.45);
SELECT ST_Line_Substring(the_geom, 0.65, 0.85);
```

And these will extract to further LINESTRINGs, the first one ranging from 15% to 45%, and the second one from 65% to 85% of the total line length [*yellow overstrike*].

ST_OffsetCurve()

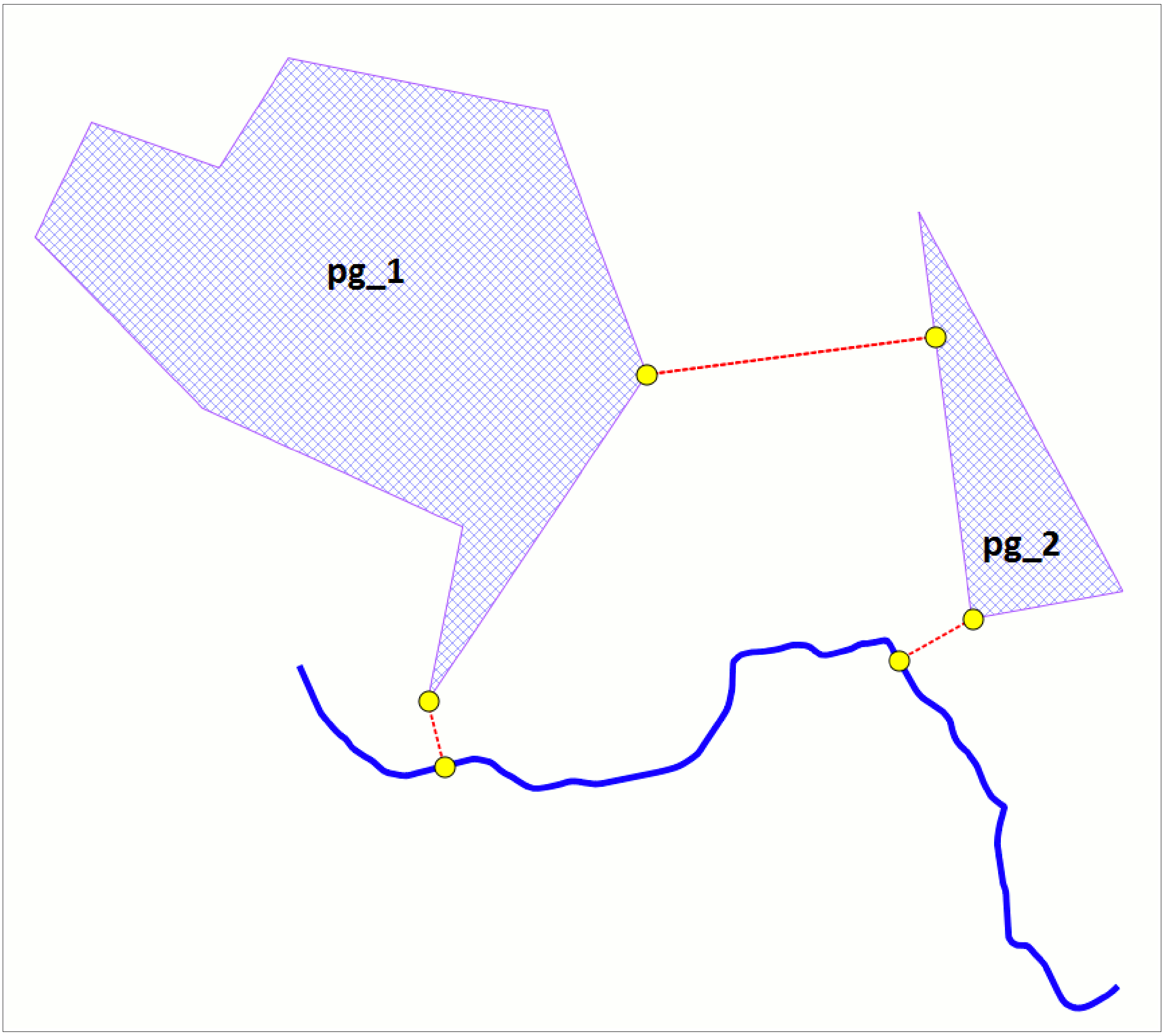


This time too we'll start from the same complex LINESTRING [*violet line*] named *the_line*.

```
SELECT ST_OffsetCurve(the_line, 10, 1);
SELECT ST_OffsetCurve(the_line, 15, 0);
```

The first function will create a *left-sided* curve with an offset factor of 10m [*orange line*]
And the second function will create a *right-sided* curve with an offset factor of 15m [*green line*]

ST_ClosestPoint() / ST_ShortestLine()



Suppose a LINESTRING named *the_line* [*blue line*] and two distinct POLYGONs named *pg_1* and *pg_2* [*violet areas*].

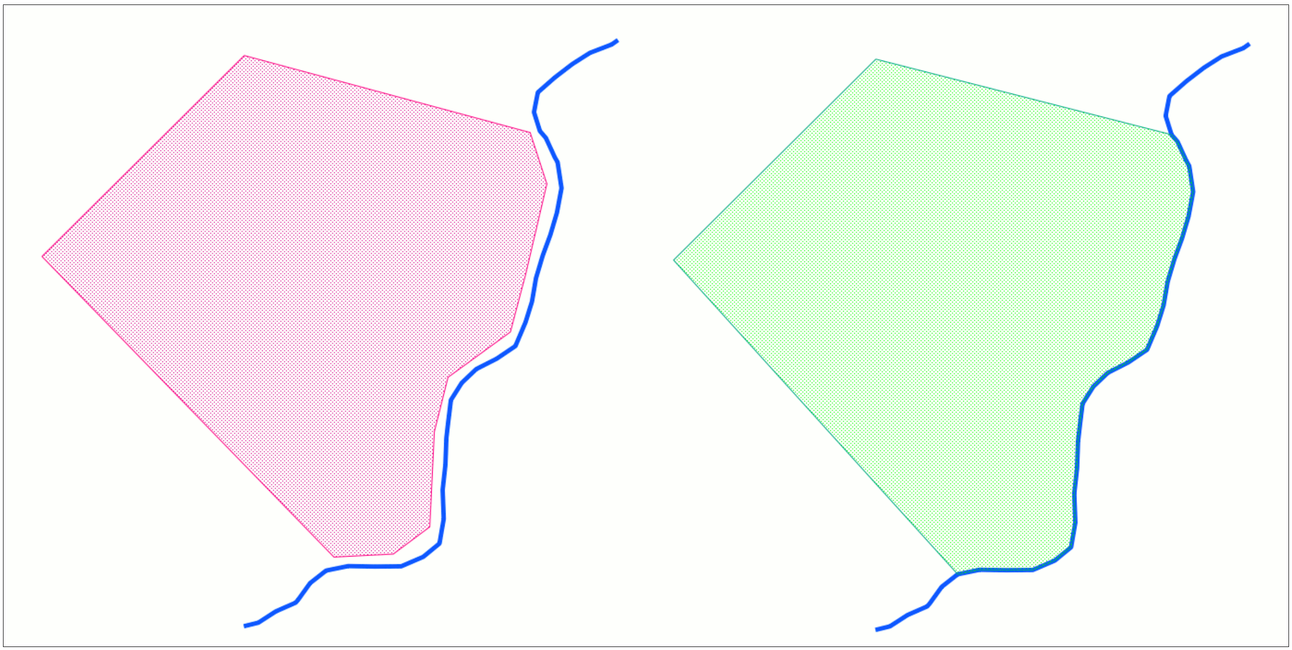
```
SELECT ST_ShortestLine(the_line, pg_1);
SELECT ST_ShortestLine(the_line, pg_2);
SELECT ST_ShortestLine(pg1, pg_2);
```

Each one of these functions will create a LINESTRING representing the minimum distance line between two arbitrary geometries [*dotted red lines*].

```
SELECT ST_ClosestPoint(the_line, pg_1);
SELECT ST_ClosestPoint(pg_1, the_line);
```

The first one will identify the POINT on *the_line* nearest to *pg_1*
And the second one will identify the POINT on *pg_1* nearest to *the_line* [*yellow dots*]
Please note: such points simply represent extremities of the corresponding minimum distance line identified by `ST_ShortestLine()`.

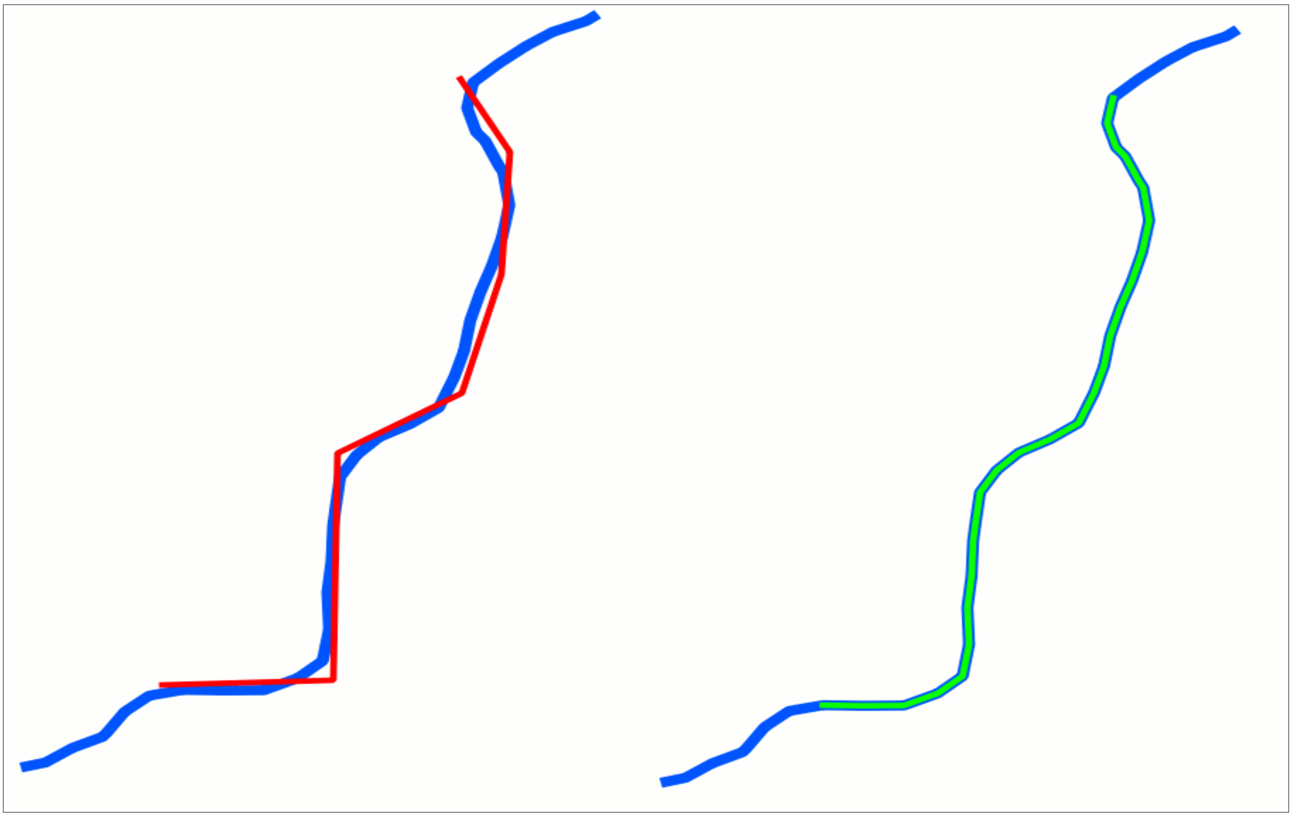
ST_Snap()



Suppose a LINESTRING named *the_line* [*blue line*] and a POLYGON named *the_polygon* [*red area*].

```
SELECT ST_Snap(the_polygon, the_line, 10.0);
```

This function will create a new POLYGON [*green area*], nicely snapped to *the_line*.



This further example shows the result of `ST_Snap()` using two LINESTRINGs

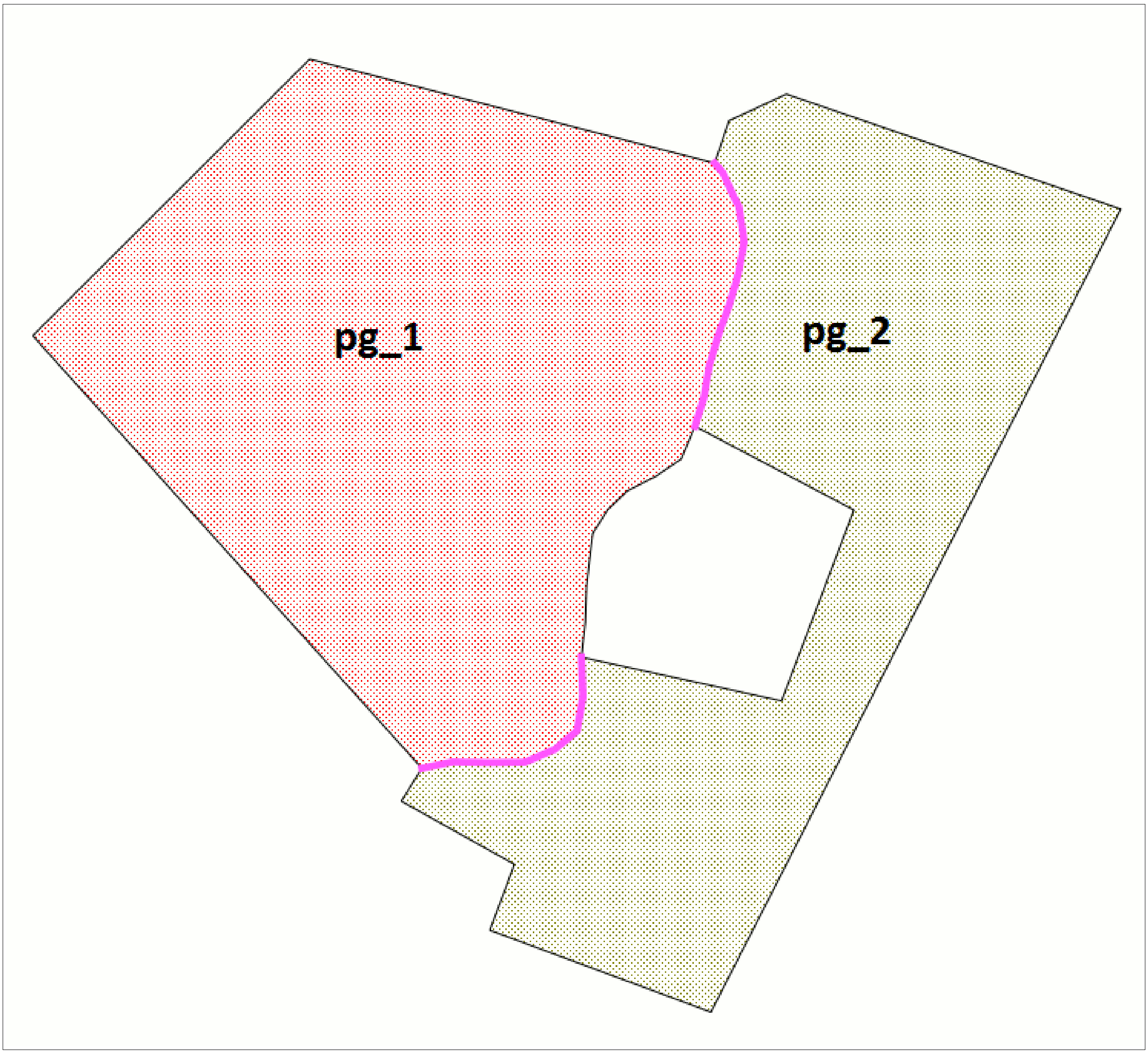
ST_Covers() / ST_CoveredBy()

We'll use again the two LINESTRINGs of the above figure: for the sake of clarity we'll name *the_blue_line* the first one, and *the_green_line* the snapped one.

```
SELECT ST_Covers(the_green_line, the_blue_line);
> 0 [false]
SELECT ST_CoveredBy(the_green_line, the_blue_line);
> 1 [true]
SELECT ST_Covers(the_blue_line, the_green_line);
> 1 [true]
SELECT ST_CoveredBy(the_blue_line, the_green_line);
> 0 [false]
```

You can use these functions to check if one Geometry fully covers (or is covered by) a second Geometry.

ST_SharedPaths()

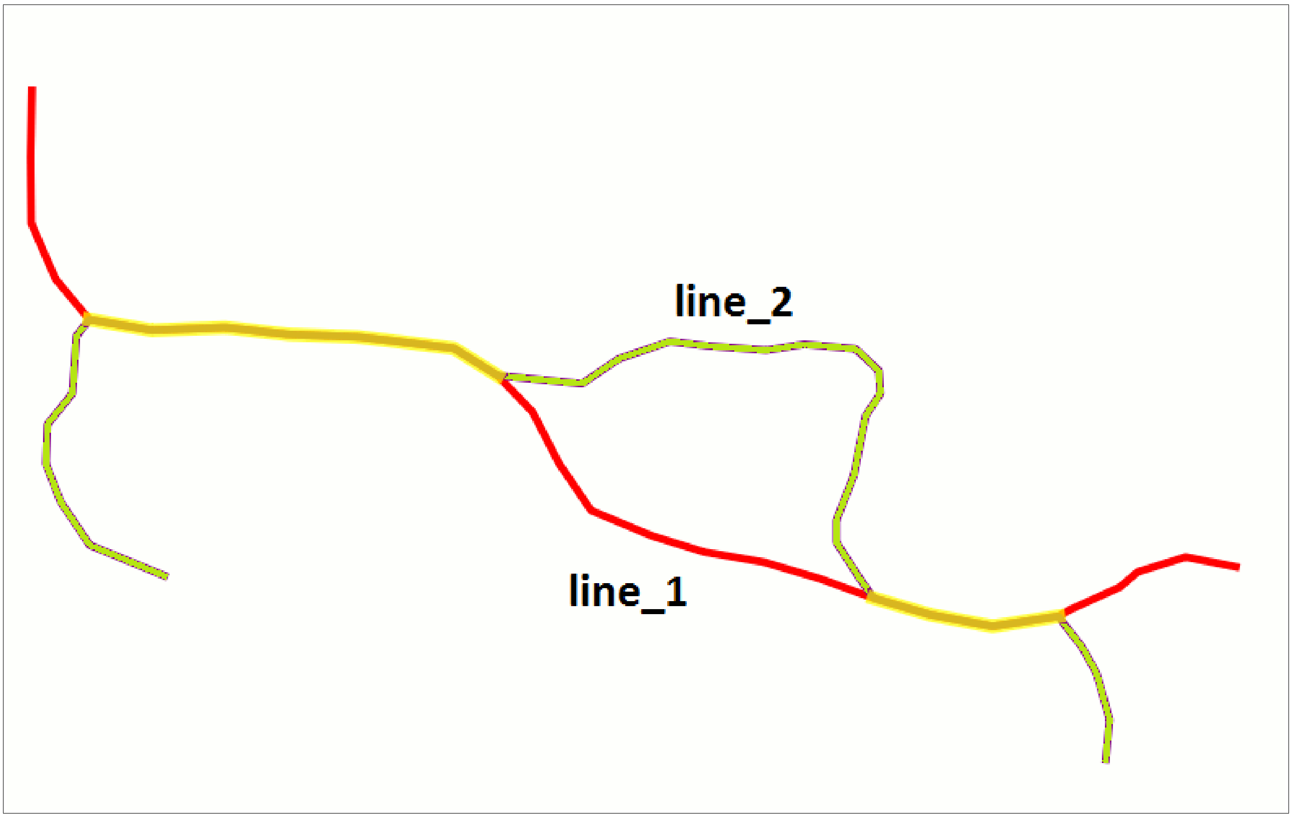


Suppose two adjacent POLYGONs respectively named *pg_1* [*red area*] and *pg_2* [*brown area*]. And imagine that a strict topological conformity exists between such polygons.

```
SELECT ST_SharedPaths(pg_1, pg_2);
```

This function will identify any *edge* portion common to both polygons [*magenta lines*].

Please note #1: the returned Geometry is constantly represented as a MULTILINESTRING, even when a single common edge has been identified.



The same operation can be applied to LINESTRINGs as well: imagine a couple of partially overlapping Linestrings respectively named *line_1* [*red*] and *line_2* [*green*].

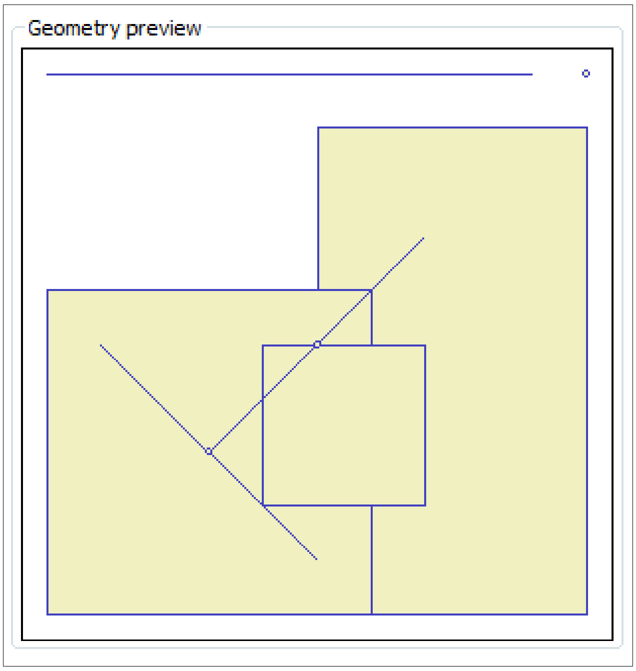
```
SELECT ST_SharedPaths(line_1, line_2);
```

Paths commons to both *line_1* and *line_2* are shown in the figure as *yellow lines*.

ST_UnaryUnion()

Suppose a quite complex GEOMETRYCOLLECTION named *the_geom:*

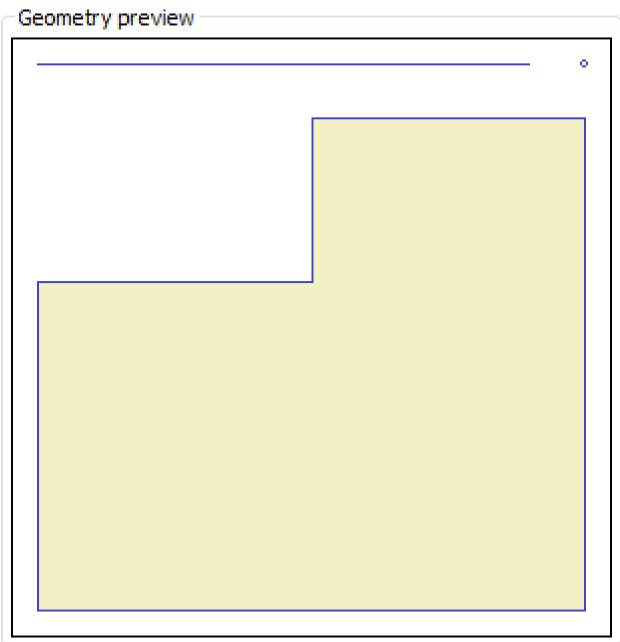
```
GEOMETRYCOLLECTION(POINT(5 5), POINT(3 3), POINT(10 10),
LINESTRING(0 10, 9 10), LINESTRING(3 3, 7 7), LINESTRING(1 5, 5 1),
POLYGON((5 0, 10 0, 10 9, 5 9, 5 0)), POLYGON((0 0, 6 0, 6 6, 0 6, 0 0)),
POLYGON((4 2, 7 2, 7 5, 4 5, 4 2)))
```



Please note: this GEOMETRYCOLLECTION is *invalid*: several items are mutually overlapping.

```
SELECT ST_AsText( ST_UnaryUnion(the_geom) );
> GEOMETRYCOLLECTION(POINT(10 10), LINESTRING(0 10, 9 10),
  POLYGON((5 0, 0 0, 0 6, 5 6, 5 9, 10 9, 10 0, 6 0, 5 0)))
```

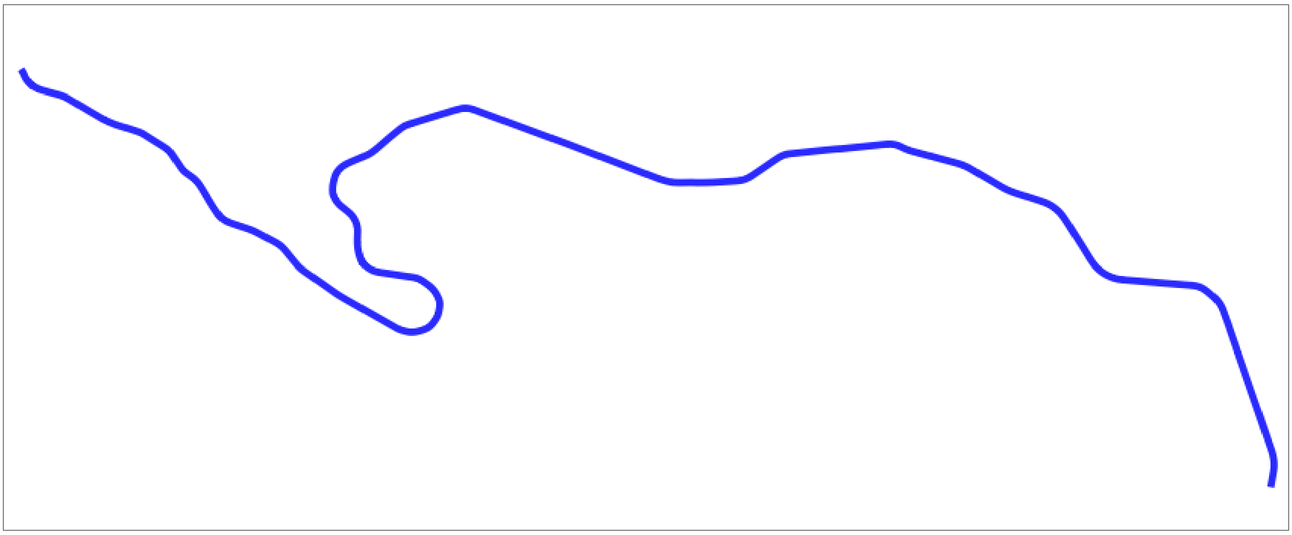
This function will recover a valid Geometry, as graphically shown in the following figure:



ST_LineMerge()

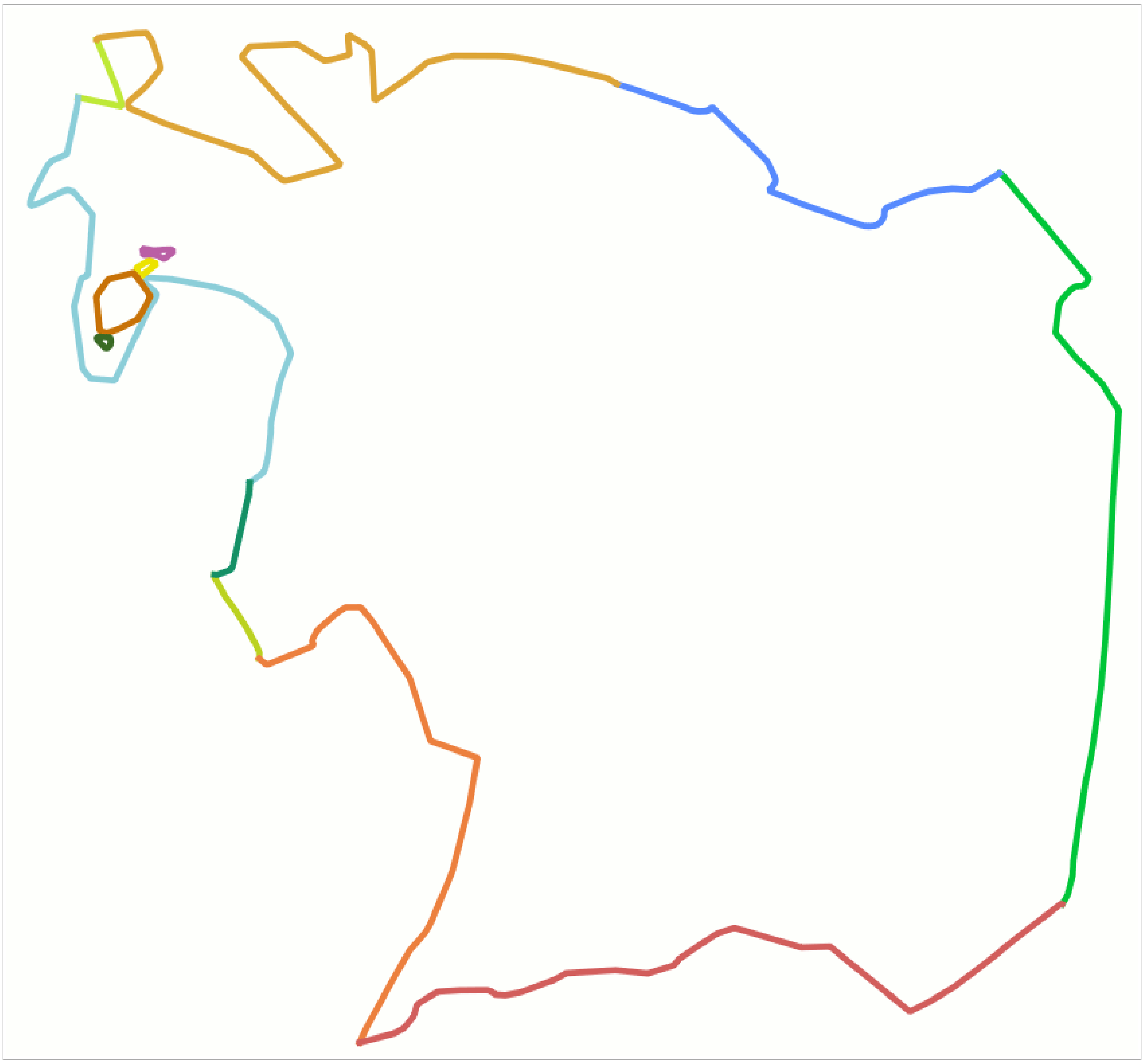
Suppose some complex road (river, railway …) represented by many and many sparse fragments.
In the above figure each single fragment is represented by a different colour.
And suppose you have been already able to put all the above fragments into a single MULTILINESTRING named *the_fragments* (*please wait for now: we'll examine this topic in a further step ...*)

```
SELECT ST_LineMerge(the_fragments);
```



All right: using `ST_LineMerge()` you'll now have a single continuous LINESTRING.

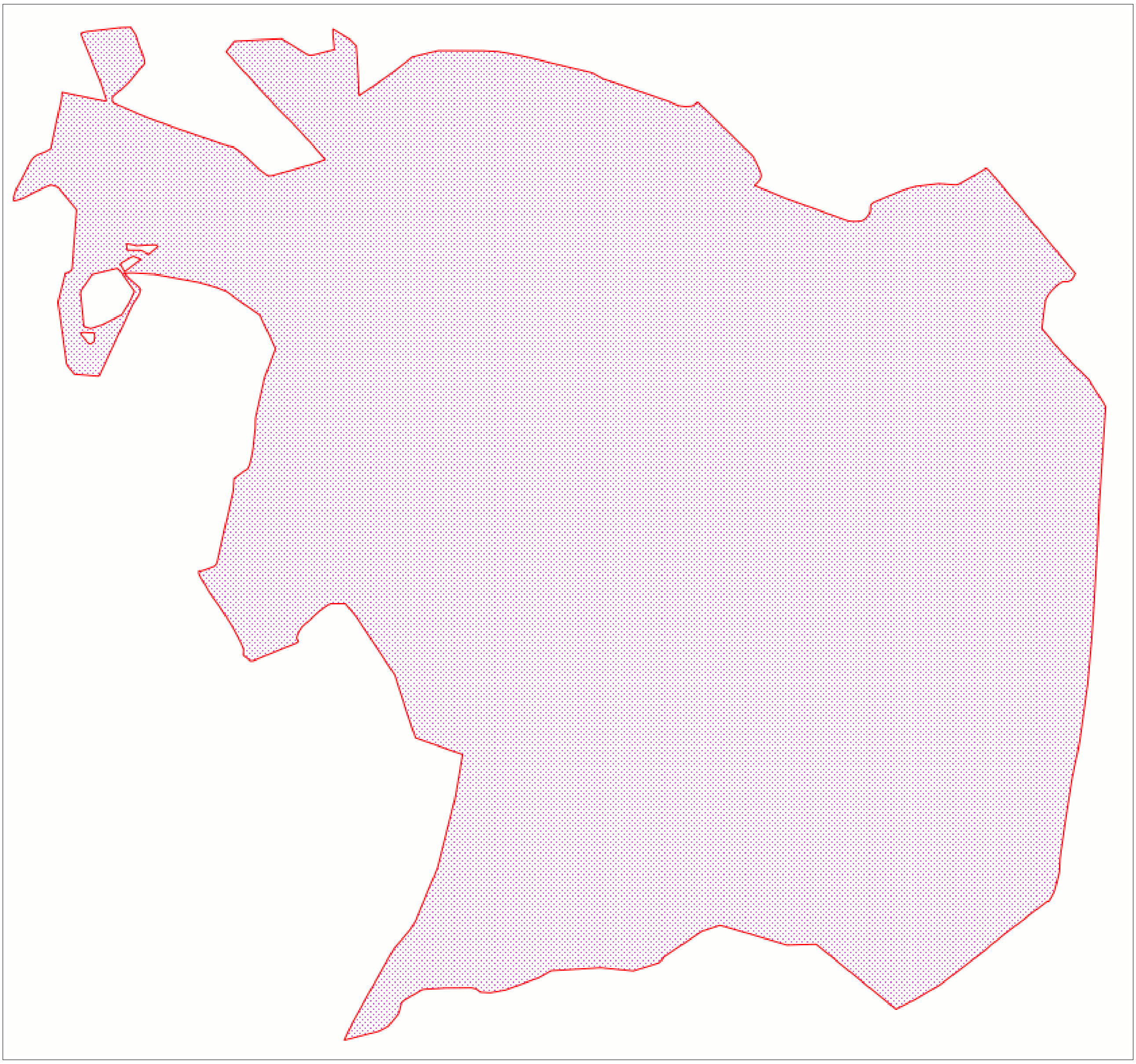
ST_BuildArea() / ST_Polygonize() / ST_Collect()



More or less, the same as above: this time too you'll start from a MULTILINESTRING. named *the_fragments* [this actually being a sparse collection of line fragments]. In the above figure each single fragment is represented by a different colour.

```
SELECT ST_BuildArea(the_fragments);
```

This function will try to reassemble a valid POLYGON / MULTIPOLYGON starting from sparse fragments.



And here is the reassembled POLYGON generated by `ST_BuildArea()`.

The `ST_Polygonize()` function performs the same identical task, but using a different syntactic approach:

```
SELECT ST_Polygonize(the_line)
FROM my_lines
WHERE polygon_id = x
GROUP BY polygon_id;
```

This one is an *aggregate function* [… `GROUP BY` …], so there is no need at all to pass a MULTILINESTRING geometry containing any required line-fragment.
Using this second approach you can simply have an ordinary table [*my_lines*], each row containing an unique line-fragment [*the_line*].

There is no conceptual difference between them: it's simply a matter of convenience and of different syntax flavors: but the *main core* is anyway one and the same.

The `ST_Collect()` function too can be successfully used on these cases, when aggregating several elementary Geometries into an unique complex Geometry is required.

```
SELECT ST_AsText(ST_Collect(
    GeomFromText('POINT(1 2)'),
    GeomFromText('POINT(3 4)')
));
> MULTIPOINT(1 2, 3 4)

SELECT ST_AsText(ST_Collect(
    GeomFromText('POINT(1 2)'),
    GeomFromText('LINESTRING(3 4, 5 6)')
));
> GEOMETRYCOLLECTION(POINT(1 2), LINESTRING(3 4, 5 6))
```

This first form accepts two arbitrary input Geometries, and return a complex Geometry representing both elementary Geometries.

```
SELECT ST_Collect(the_line)
FROM my_lines
WHERE polygon_id = x
GROUP BY polygon_id;
```

This second form is an *aggregate function* instead: and will return a complex Geometry representing any elementary Geometry found on the underlying aggregate target.

```
SELECT ST_Polygonize(the_line)
FROM my_lines
WHERE polygon_id = x
GROUP BY polygon_id;

SELECT ST_BuildArea(ST_Collect(the_line))
FROM my_lines
WHERE polygon_id = x
GROUP BY polygon_id;
```

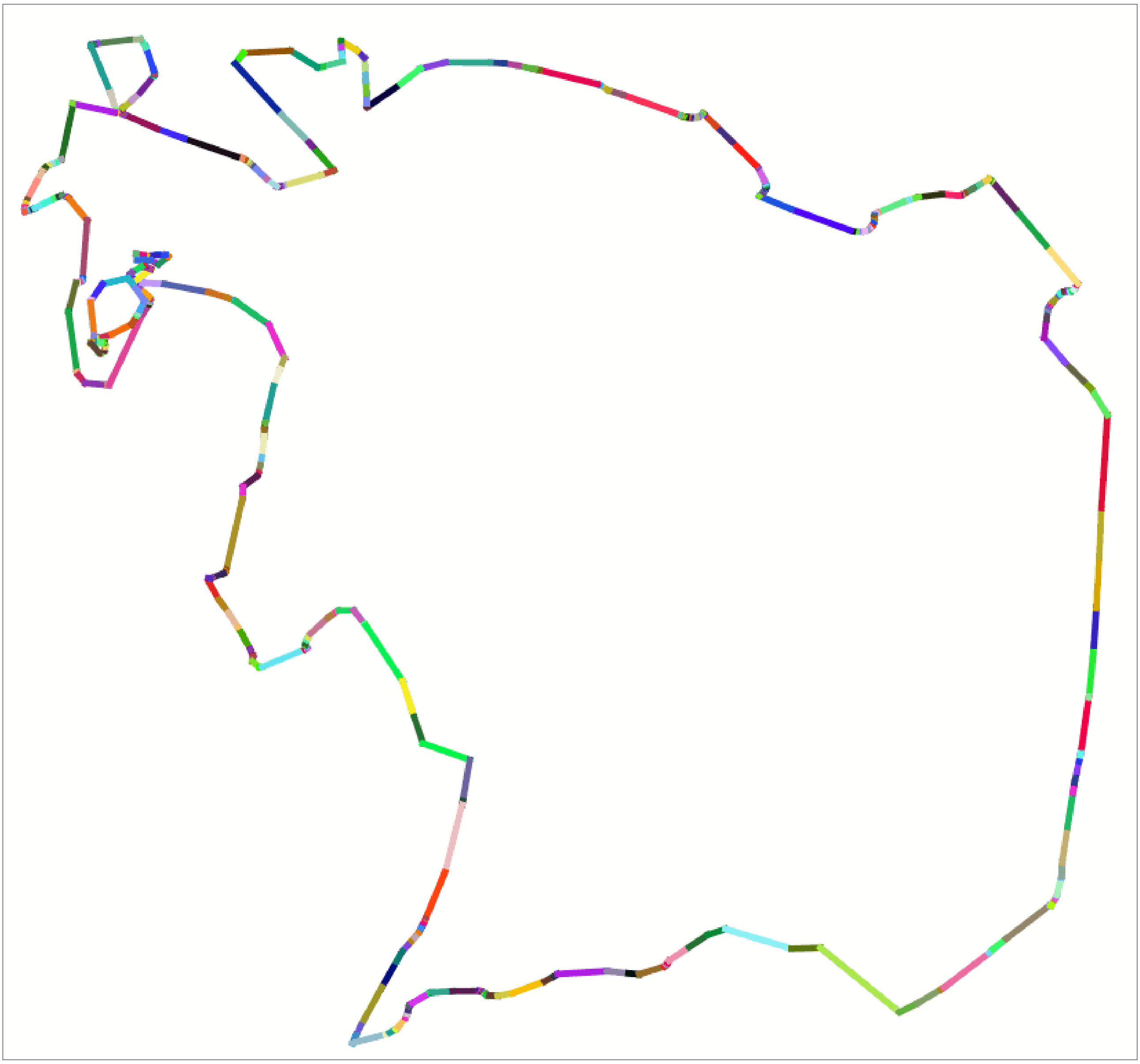
Accordingly to all the above considerations, both SQL queries performs the same identical task. They only apparently are different; but in substantial terms they are absolutely equivalent.

ST_DissolvePoints() / ST_DissolveSegments()

We'll start again using the same POLYGON of the previous example, naming it as *the_polygon*

```
SELECT ST_DissolvePoints(the_polygon);
```

This function will *dissolve* any arbitrary Geometry into a MULTIPOINT: any POINT will remain unaffected, but any LINESTRING or RING will simply be represented by its Vertices.



```
SELECT ST_DissolveSegments(the_polygon);
```

And this second function will *dissolve* any arbitrary Geometry into a MULTILINESTRING or GEOMETRYCOLLECTION: any POINT will remain unaffected, but any LINESTRING or RING will be then represented by simple segments (*each one of them being represented by a different colour in the above figure*).

Please note: dissolving into segments some broken (*invalid*) POLYGON/MULTIPOLYGON, and then calling `ST_BuildArea()` may be a good approach to recover a valid Geometry.

ST_CollectionExtract()

Suppose a quite complex GEOMETRYCOLLECTION named *the_geom:*

```
GEOMETRYCOLLECTION(POINT(105 105), POINT(103 103), POINT(102 102),
LINESTRING(0 10, 9 10), LINESTRING(30 30, 37 37), LINESTRING(51 55, 55 51),
POLYGON((75 70, 80 70, 80 79, 75 79, 75 70)))
```

You can invoke the `ST_CollectionExtract()` function in order to extract elementary Geometries from the Collection by homogeneous type.

```
SELECT ST_AsText( ST_CollectionExtract(the_geom, 1) );
> MULTIPOINT(105 105, 103 103, 102 102)
```

Please note: the argument ***1*** identifies the POINT type.

```
SELECT ST_AsText( ST_CollectionExtract(the_geom, 2) );
> MULTILINESTRING((0 10, 9 10), (30 30, 37 37), (51 55, 55 51))
```

Please note: the argument ***2*** identifies the LINESTRING type.

```
SELECT ST_AsText( ST_CollectionExtract(the_geom, 3) );
> MULTIPOLYGON(((75 70, 80 70, 80 79, 75 79, 75 70)))
```

Please note: the argument ***3*** identifies the POLYGON type.

all this new cool features ...

... will be released ASAP

SpatiaLite v.3.0 is coming !!!