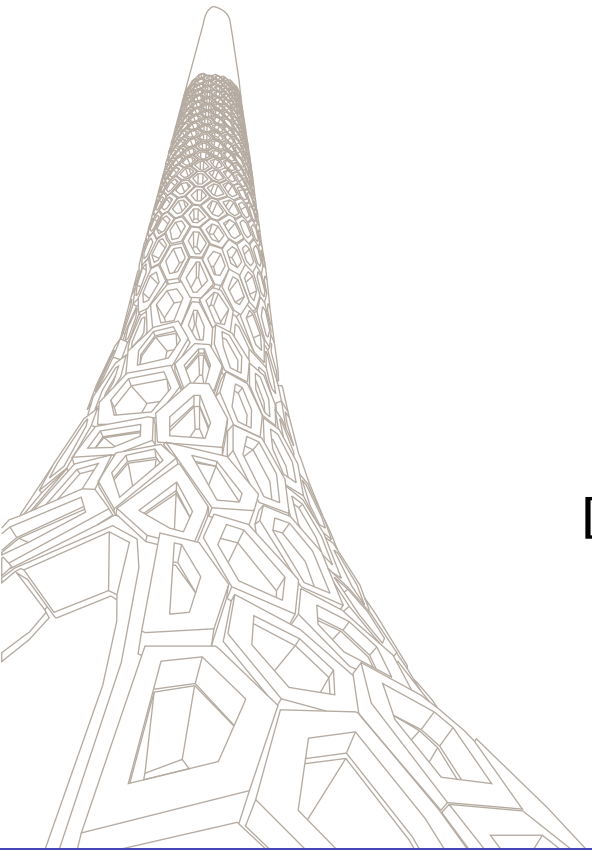
An introduction to MetricGraph

Graph construction and data handling

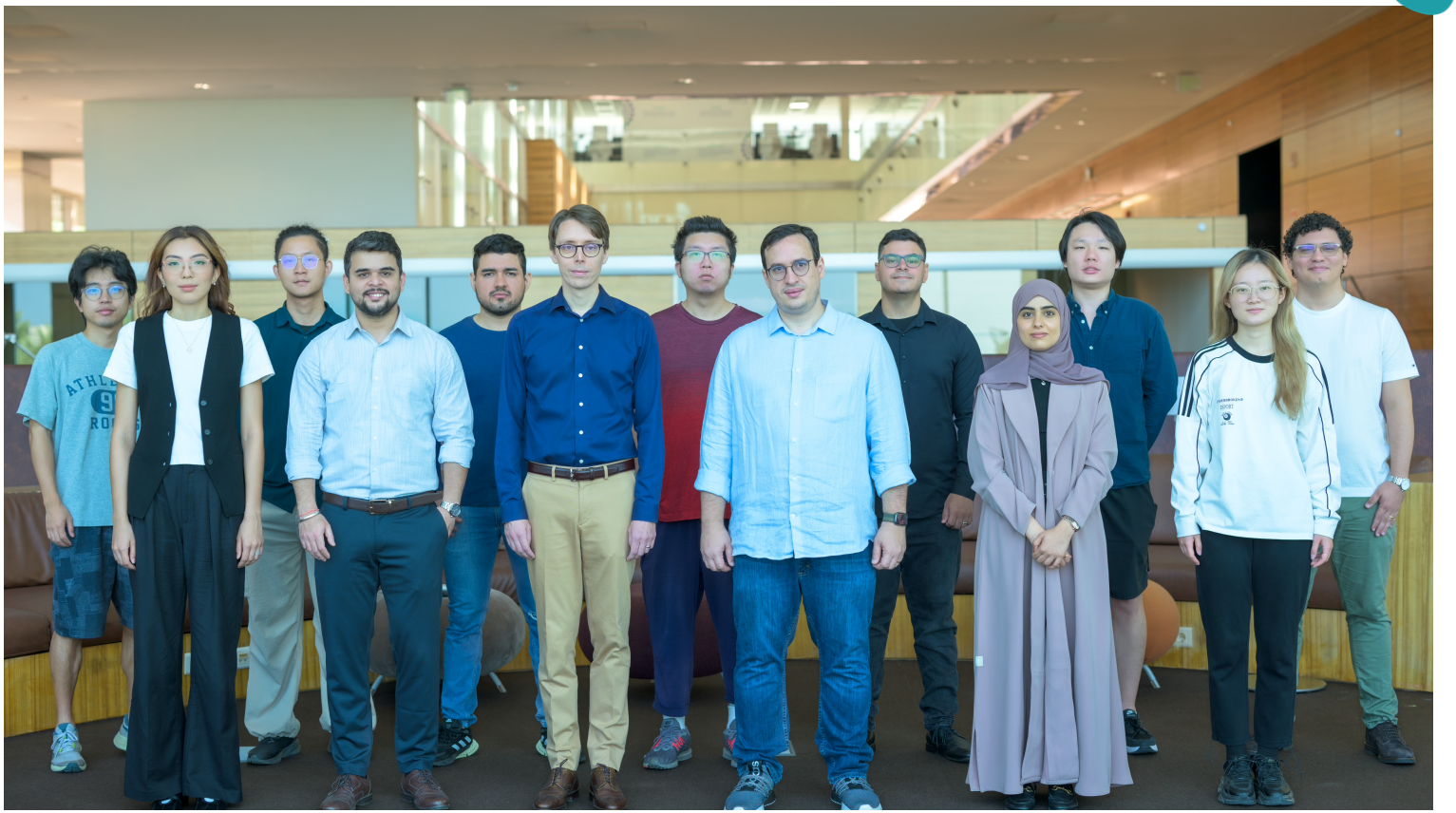


David Bolin and Alexandre Simas

Glasgow 2025

Stochastic Processes and Applied Statistics

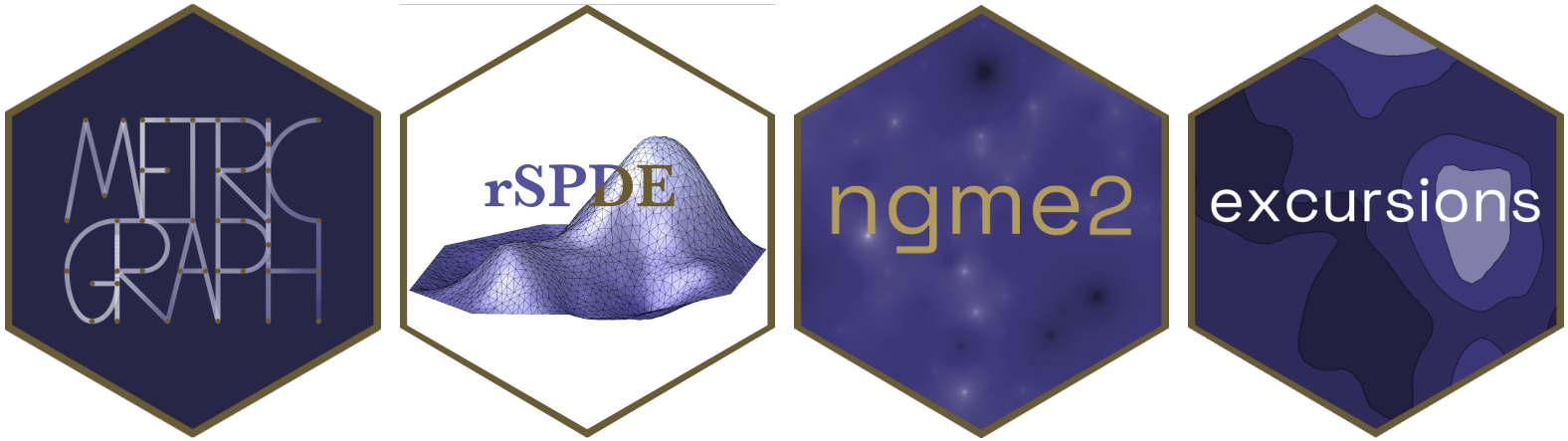
Stochastic Process & Applied Statistics



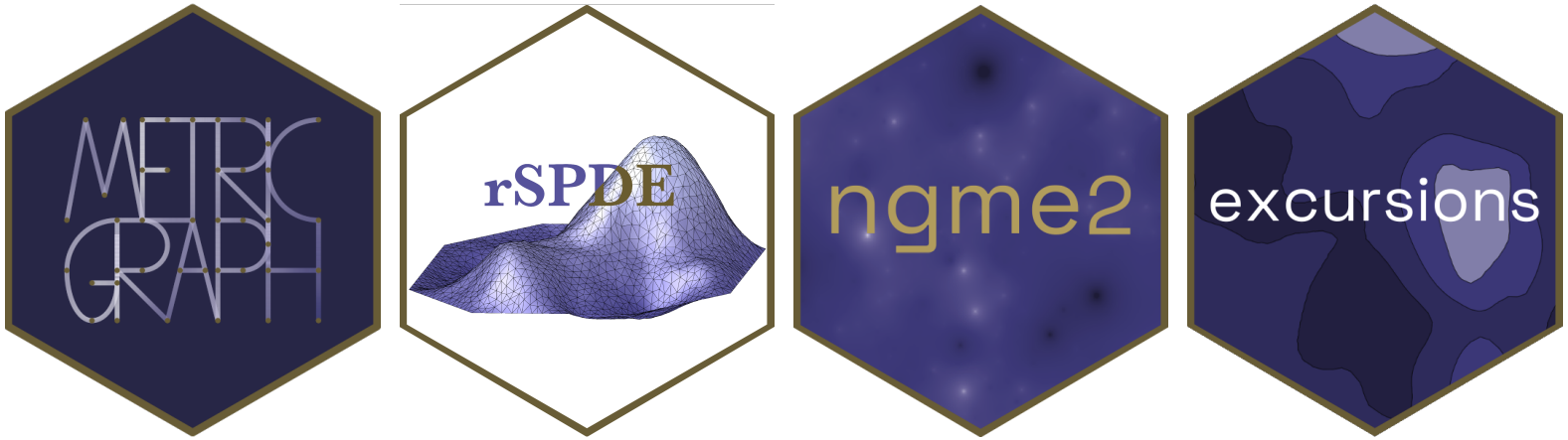
Main research focus

Statistical methods based on SPDEs, in particular modeling and inference using non-Gaussian and fractional-order SPDEs. In the interface between statistics, probability, applied mathematics and numerical analysis.

INLA-related packages

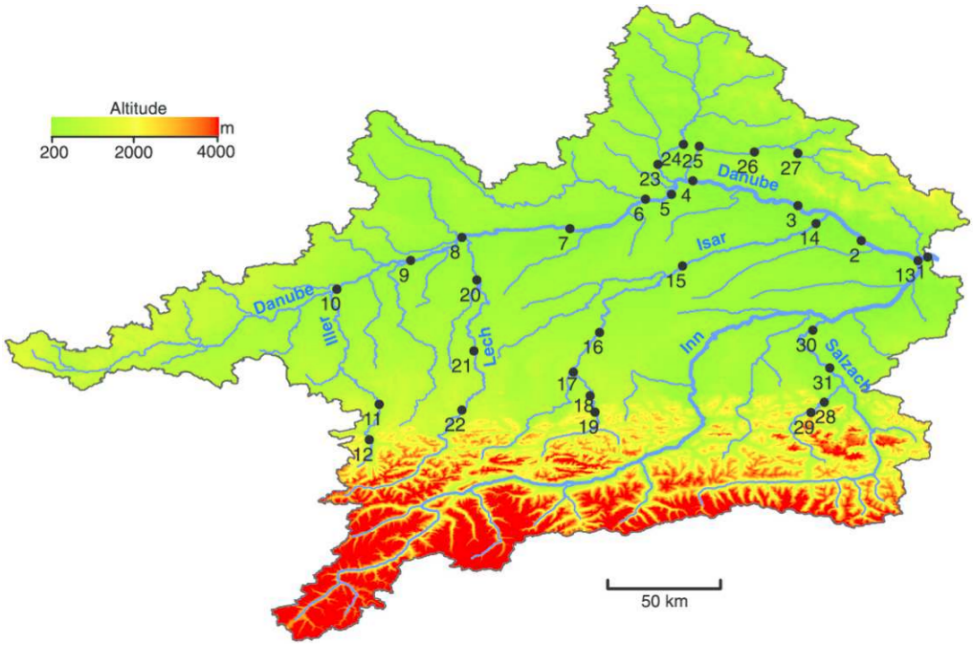


INLA-related packages



Looking for students and postdocs, let me know if you are interested.

Spatial data on networks



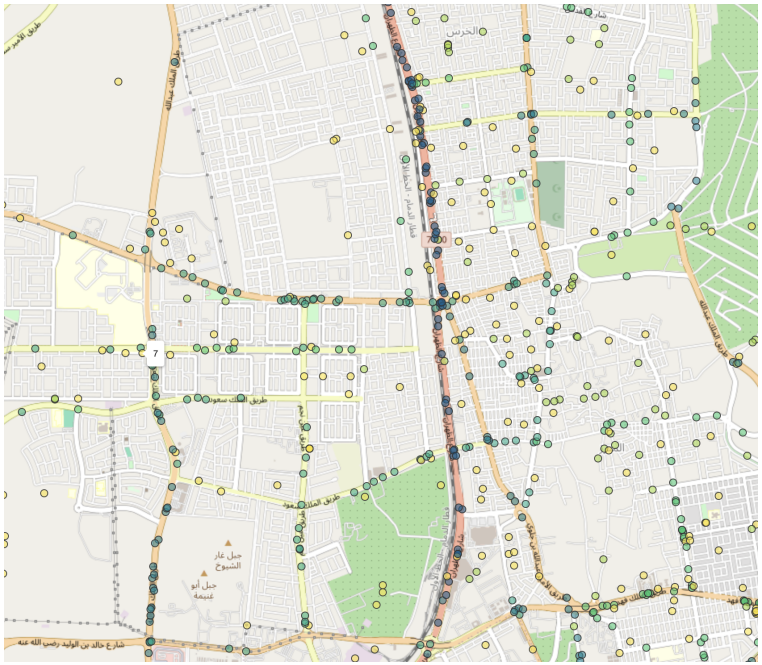


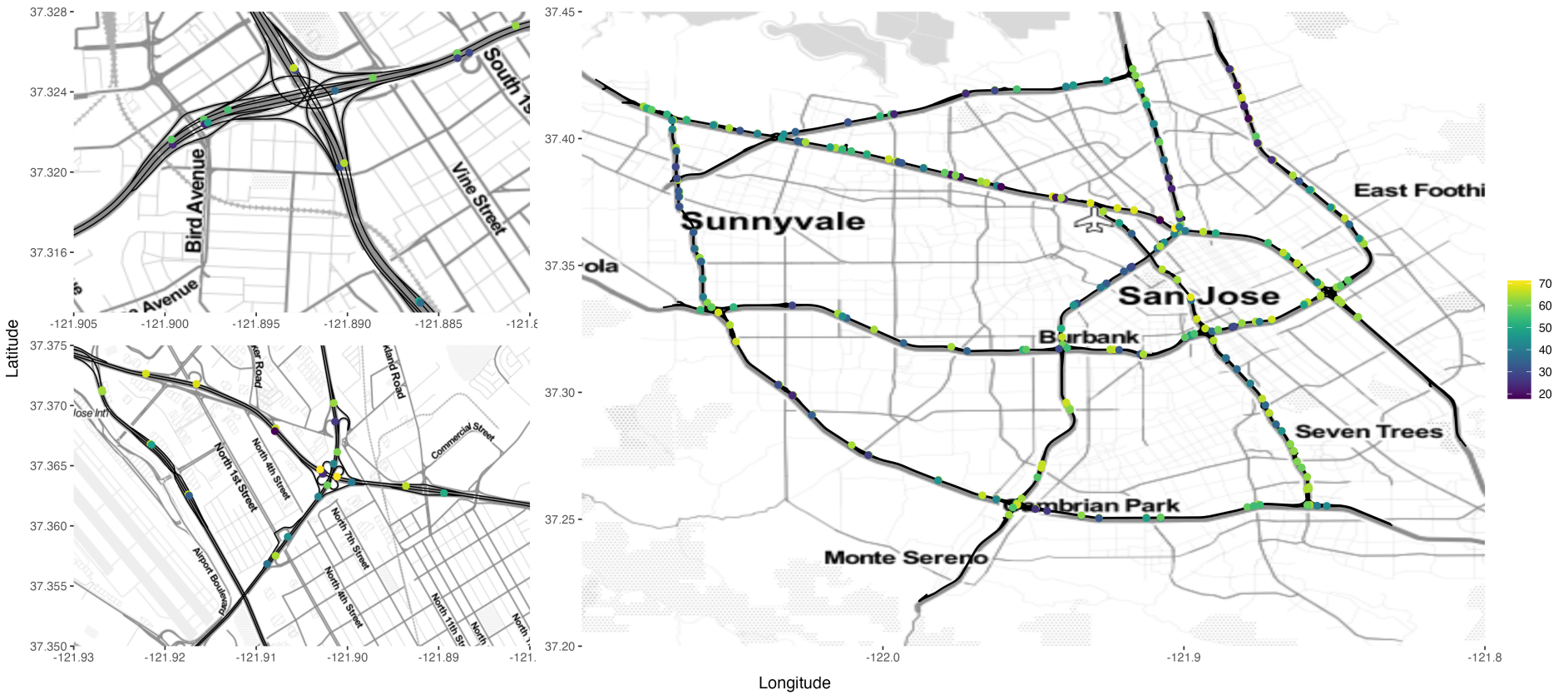
Figure 1: Danube river network and traffic accident locations in Al-Ahsa.

- Distance should be measured on the network.
- We are interested in statistical modeling data on networks.
- The processes should be defined in continuous space.
- A starting point is Gaussian processes.

Covariance-based approaches

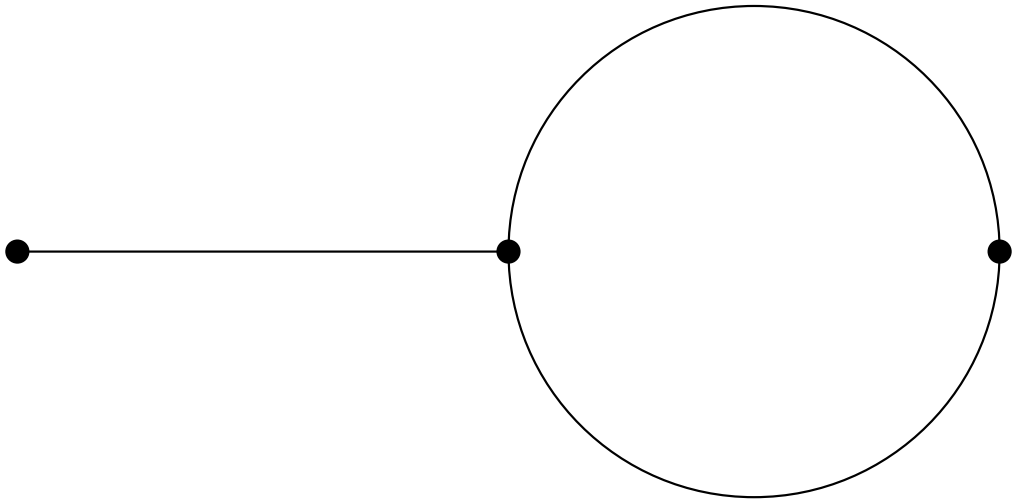
- Covariance-based stream network models (SSN)
  - limited to stream networks (no loops)
  - SSN package is popular
- Isotropic covariance-based models
  - Only works on graphs with Euclidean edges (e.g., no multiple edges)
  - No way to define differentiable Gaussian processes
  - Is isotropy reasonable on these domains?
- Computationally expensive, not compatible with INLA

Barrier models



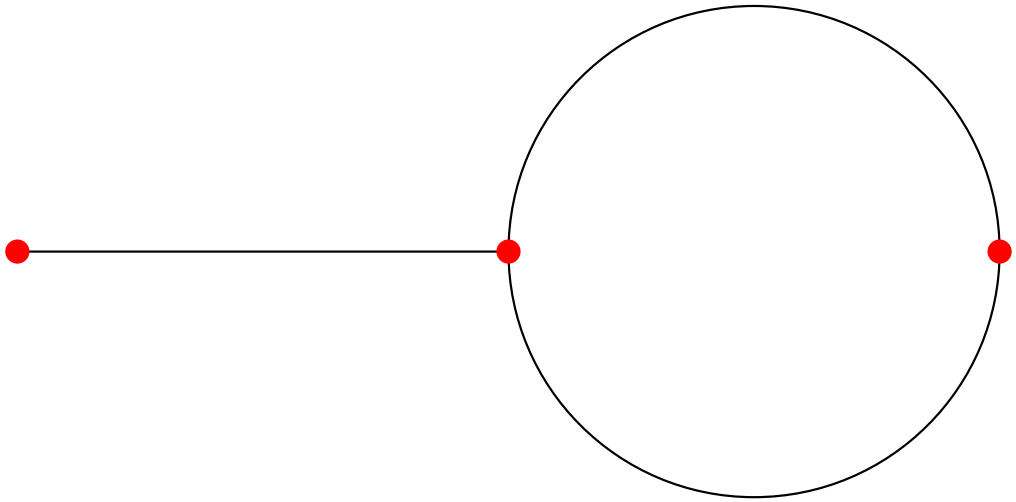
- The existing approach in INLA
- Based on triangulating the entire spatial domain (does not scale well)
- Cannot handle bridges or tunnels

Definition of metric graphs



A compact metric graph Γ consists of

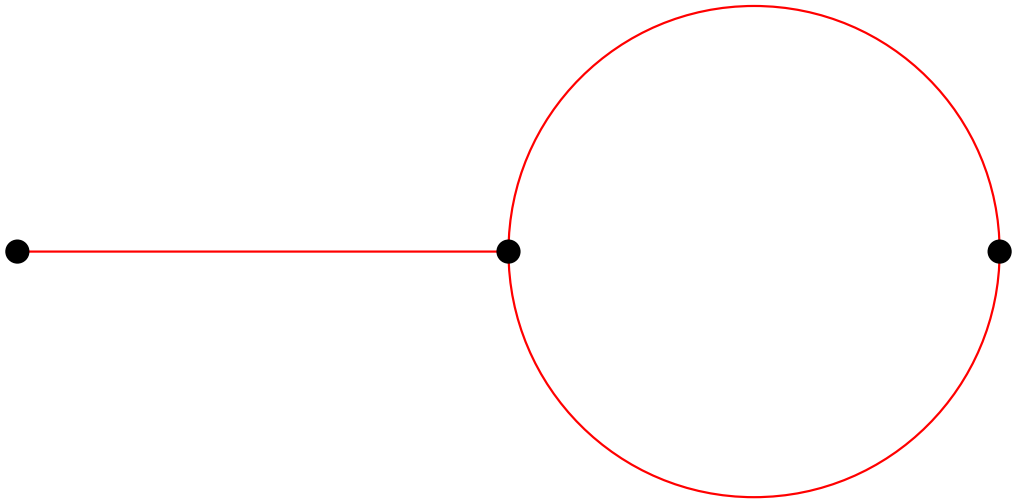
Definition of metric graphs



A compact metric graph Γ consists of

- A finite set of vertices $\mathcal{V} = \{v_i\}$.

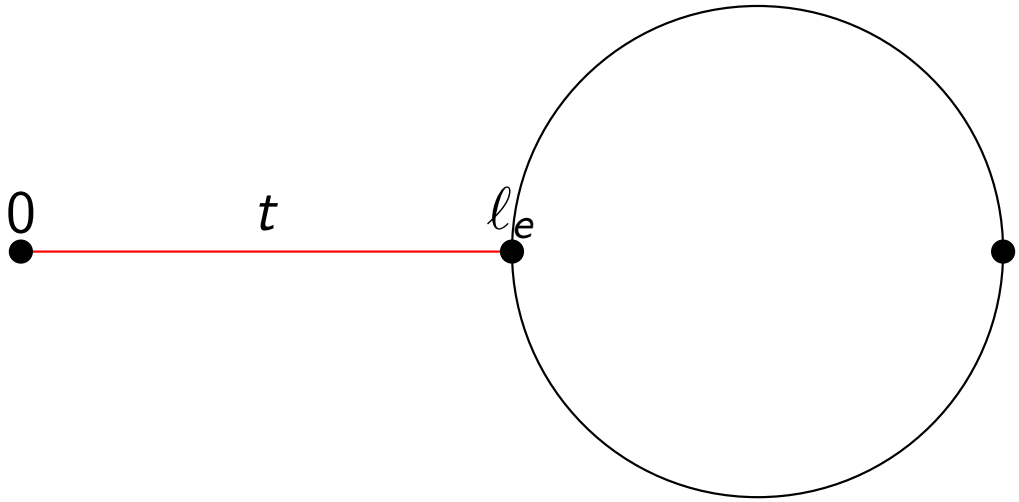
Definition of metric graphs



A compact metric graph Γ consists of

- A finite set of vertices $\mathcal{V} = \{v_i\}$.
- A finite set of edges $\mathcal{E} = \{e_j\}$.

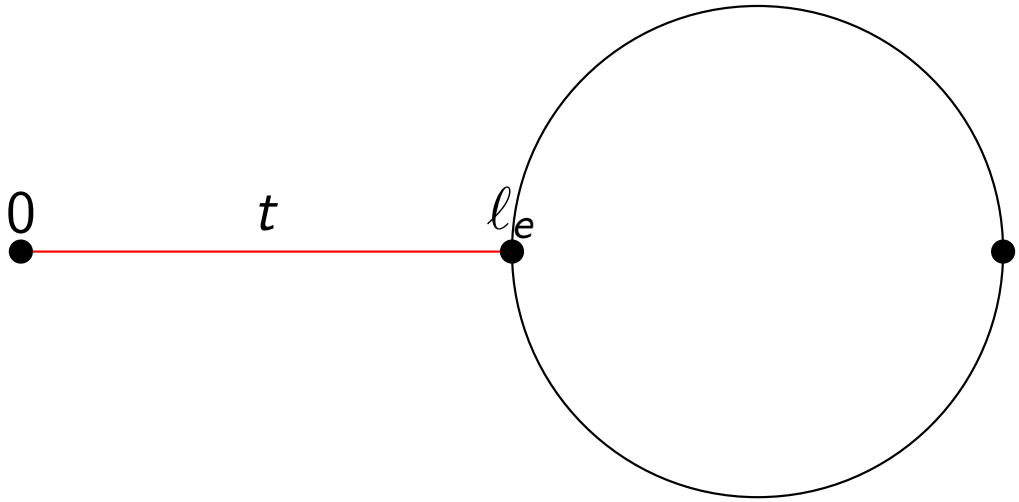
Definition of metric graphs



A compact metric graph Γ consists of

- A finite set of vertices $\mathcal{V} = \{v_i\}$.
- A finite set of edges $\mathcal{E} = \{e_j\}$.
- Each edge is a curve parameterized by arc length in $[0, \ell_e]$ with $\ell_e < \infty$.

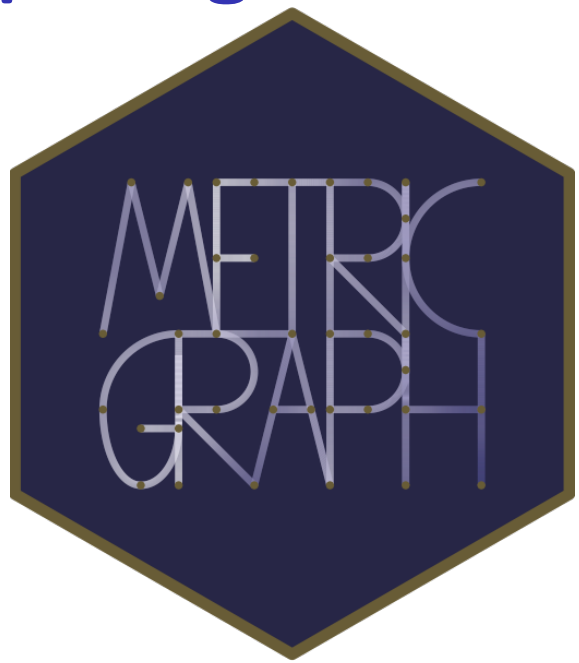
Definition of metric graphs



A compact metric graph Γ consists of

- A finite set of vertices $\mathcal{V} = \{v_i\}$.
- A finite set of edges $\mathcal{E} = \{e_j\}$.
- Each edge is a curve parameterized by arc length in $[0, \ell_e]$ with $\ell_e < \infty$.
- We equip the graph with the shortest path distance $d(x, y)$ as metric.

The `MetricGraph` package



- Contains functionality for working with data on metric graphs.
- Implements various Gaussian random fields.
- Has interfaces to `R-INLA` and `inlabru`.
- Available on CRAN. Homepage: https://davidbolin.github.io/MetricGraph/

Theory references

- Bolin, Simas, Wallin (2024). Gaussian Whittle-Matérn fields on metric graphs, *Bernoulli*
- Bolin, Kumar, Kovacs, Simas (2024). Regularity and numerical approximation of fractional elliptic differential equations on compact metric graphs, *MathComp*
- Bolin, Simas, Wallin (2025). Markov properties of Gaussian random fields on metric graphs, *Bernoulli*
- Bolin, Simas, Wallin (2025+). An explicit link between graphical models and Gaussian Markov random fields on metric graphs, *Stochastic Processes & their applications, revised*
- Bolin, Simas, Wallin (2025+). Statistical inference for Gaussian Whittle-Matérn fields on metric graphs, *Journal of the Royal Statistical Society, Series B, in revision*
- Bolin, Riera-Segura, Simas (2025+). A new class of non-stationary Gaussian processes with general smoothness on metric graphs, *arXiv preprint*
- Bolin, Saduakhas, Simas (2025+). Log-Gaussian Cox processes on metric graphs, *arXiv preprint*

Outline

First hour:

- Construction of metric graphs
- Working with data on metric graphs
- Vizualization

Secont hour:

- Statistical modeling

Outline

First hour:

- Construction of metric graphs
- Working with data on metric graphs
- Vizualization

Secont hour:

- Statistical modeling

At the heart of `MetricGraph` is an `R6` class `metric_graph` which is used to define metric graphs, store data, visualize graphs and data.

The `metric_graph` function

A metric graph can be constructed in different ways. The first is to specify all edges in the graph as a list object, where each entry is a matrix.

```
edge1 <- rbind(c(0,0),c(1,0))
edge2 <- rbind(c(0,0),c(0,1))
edge3 <- rbind(c(0,1),c(-1,1))
theta <- seq(from=pi,to=3*pi/2,length.out = 50)
edge4 <- cbind(sin(theta),1+ cos(theta))
edges = list(edge1, edge2, edge3, edge4)
```

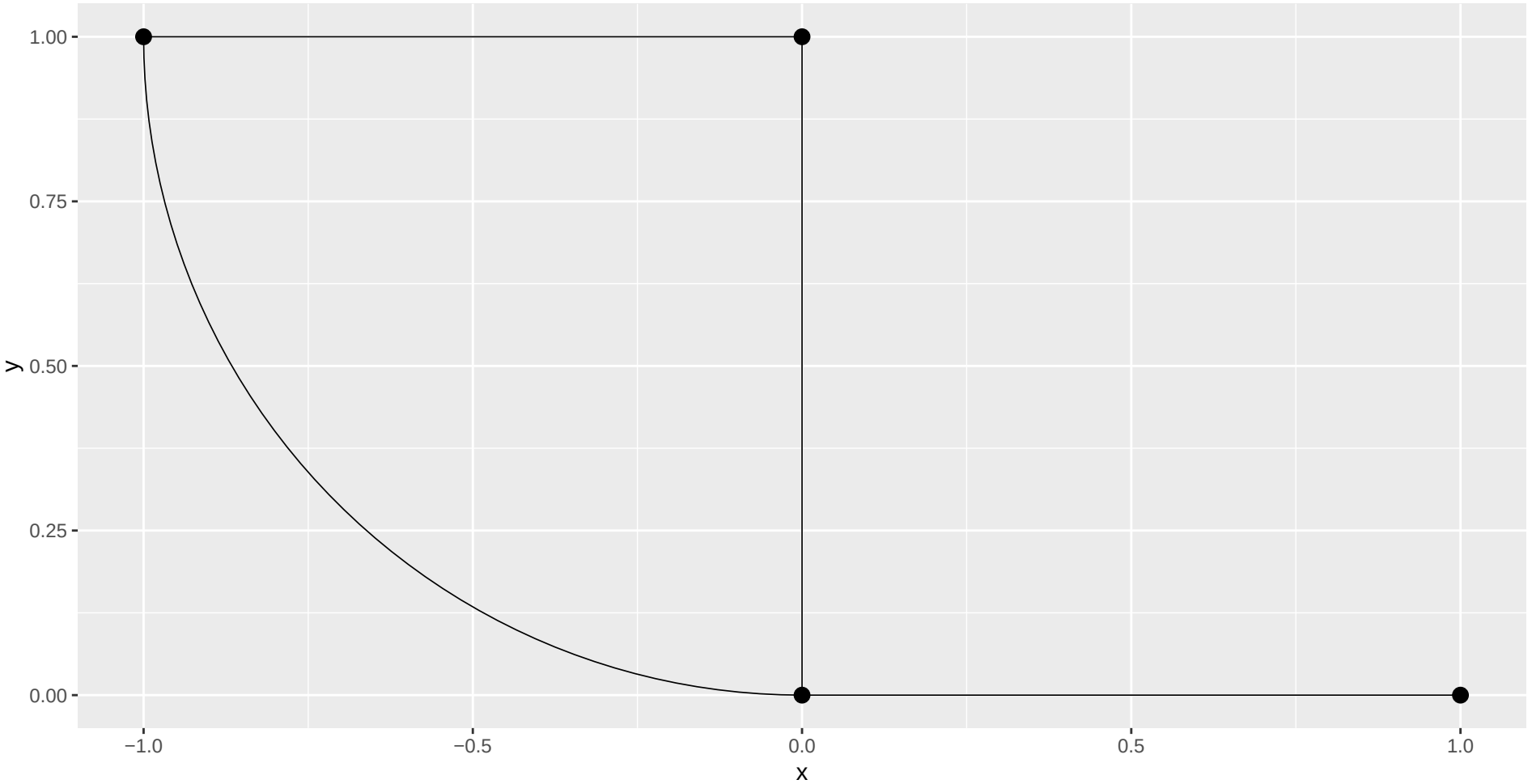
We can now create the graph based on the edges object as follows

```
graph <- metric_graph$new(edges = edges)
```

Visualization using ggplot2

To vizualize the graph using ggplot2, we do

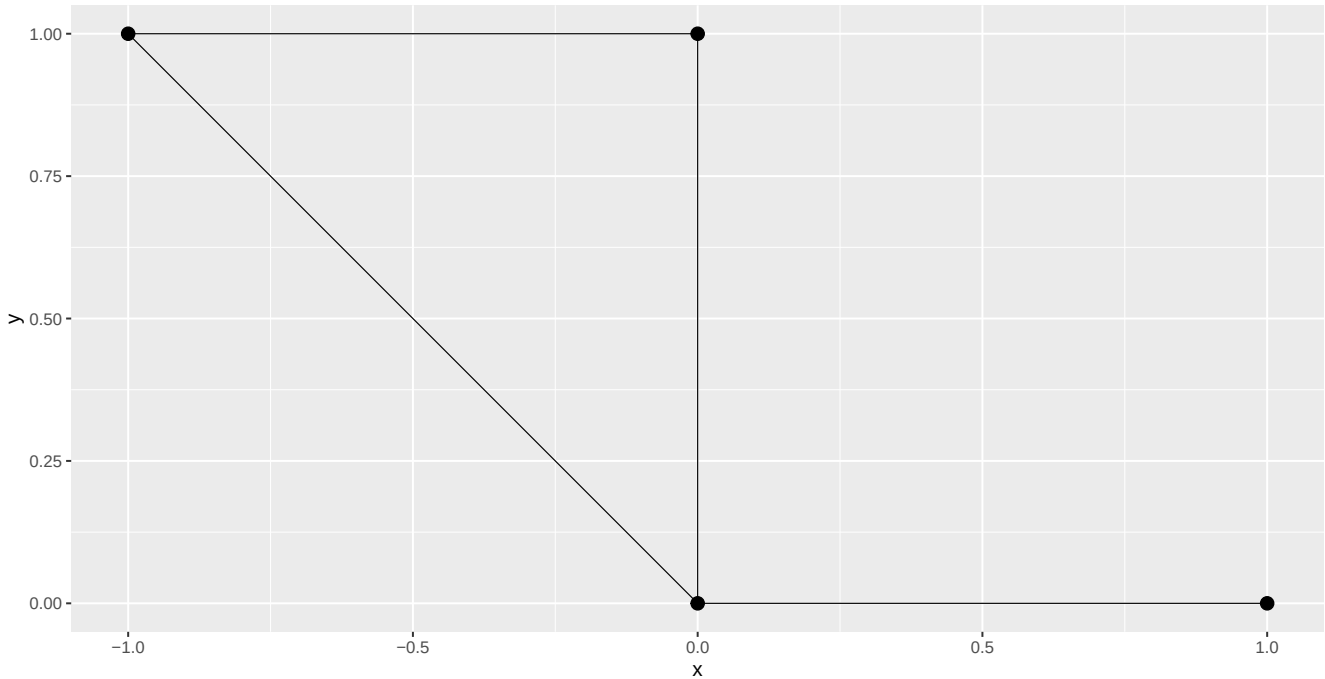
```
graph$plot()
```



Alternative graph creation methods

A linear network can be specified through the matrices `V` and `E` that specify the locations (in Euclidean space) of the vertices and the edges.

```
V <- rbind(c(0, 0), c(1, 0), c(0, 1), c(-1, 1))
E <- rbind(c(1, 2), c(1, 3), c(3, 4), c(4, 1))
graph2 <- metric_graph$new(V = V, E = E)
graph2$plot()
```



Alternative graph creation methods

A third option is to create a graph from a SpatialLines object:

```
library(sp)
line1 <- Line(rbind(c(0,0),c(1,0)))
line2 <- Line(rbind(c(0,0),c(0,1)))
line3 <- Line(rbind(c(0,1),c(-1,1)))
line4 <- Line(cbind(sin(theta),1+ cos(theta)))
lines <- sp::SpatialLines(list(Lines(list(line1),ID="1"),
                               Lines(list(line2),ID="2"),
                               Lines(list(line3),ID="3"),
                               Lines(list(line4),ID="4")))
graph <- metric_graph$new(edges = lines)
```

Alternative graph creation methods

A fourth option is to create a graph from a `MULTILINESTRING` object:

```
library(sf)
line1 <- st_linestring(rbind(c(0,0),c(1,0)))
line2 <- st_linestring(rbind(c(0,0),c(0,1)))
line3 <- st_linestring(rbind(c(0,1),c(-1,1)))
line4 <- st_linestring(cbind(sin(theta),1+ cos(theta)))
lines <- st_multilinestring(list(line1, line2,
                                 line3, line4))

graph <- metric_graph$new(edges = lines)
```

Visualization using `plotly`

To vizualize the graph using `plotly`, we do

```
graph$plot(type = "plotly")
```

Exercise

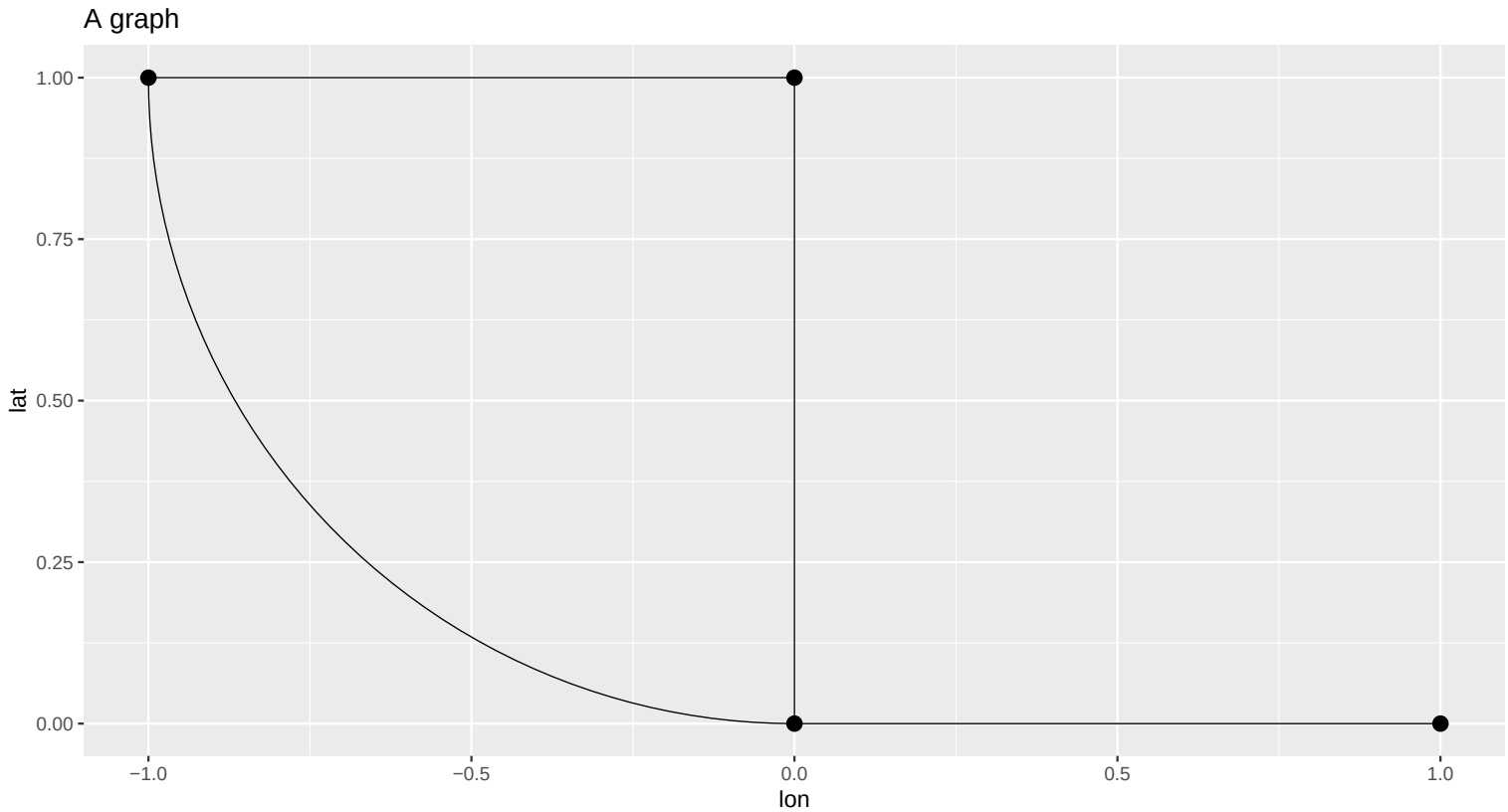
Try creating a small graph and plotting it using plotly.
If you cannot think of a graph, you can simply use the package logo:

```
graph <- metric_graph$new()
```

Working with plots

By default, the `plot` method returns a `ggplot2` object:

```
p <- graph$plot()
p + ggplot2::labs(title = "A graph", x = "lon", y = "lat")
```



Similarly, a `plotly` object is returned when `type = plotly` is set.

Tolerances

The constructor of the graph has one argument, `perform_merges` and an additional argument `tolerance` which is used for connecting edges that are close in Euclidean space.

The tolerance argument is given as a list with three elements:

`vertex_vertex` vertices that are closer than this number are merged (the default value is 1e-7)

`vertex_edge` if a vertex at the end of one edge is closer than this number to another edge, this vertex is connected to that edge (the default value is 1e-7)

`edge_edge` if two edges at some point are closer than this number, a new vertex is added at that point and the two edges are connected (the default value is 0 which means that the option is not used)

Exercise (1)

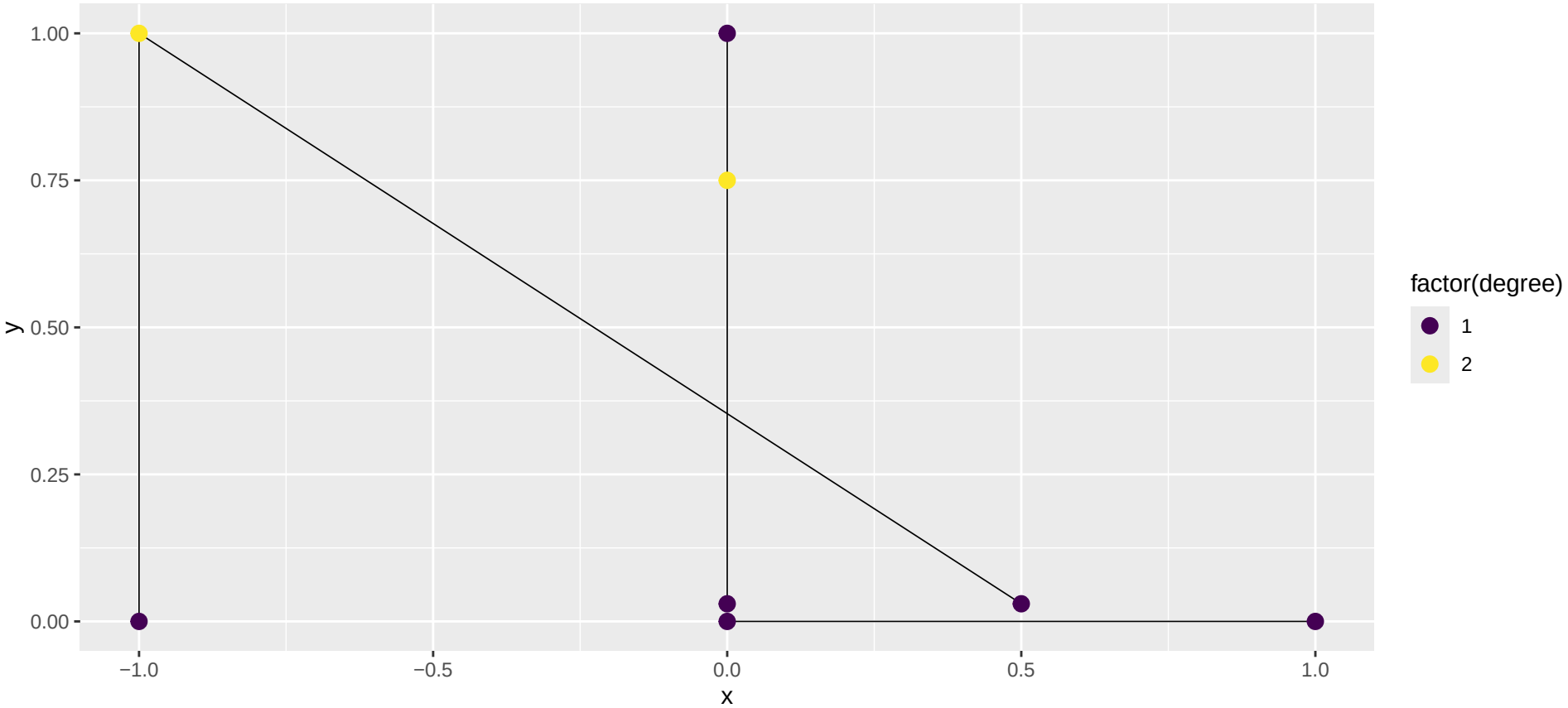
Consider the following setup and see how changing the `tolerance` argument changes the graph.

```
edge1 <- rbind(c(0,0),c(1,0))
edge2 <- rbind(c(0,0.03),c(0,0.75))
edge3 <- rbind(c(-1,1),c(0.5,0.03))
edge4 <- rbind(c(0,0.75), c(0,1))
edge5 <- rbind(c(-1,0), c(-1, 0.9995))
edges =  list(edge1, edge2, edge3, edge4, edge5)
graph3 <- metric_graph$new(edges = edges,
                           perform_merges = TRUE)
```

Exercise (2)

Specifying `degree = TRUE` in the plot function shows the degrees of the vertices.

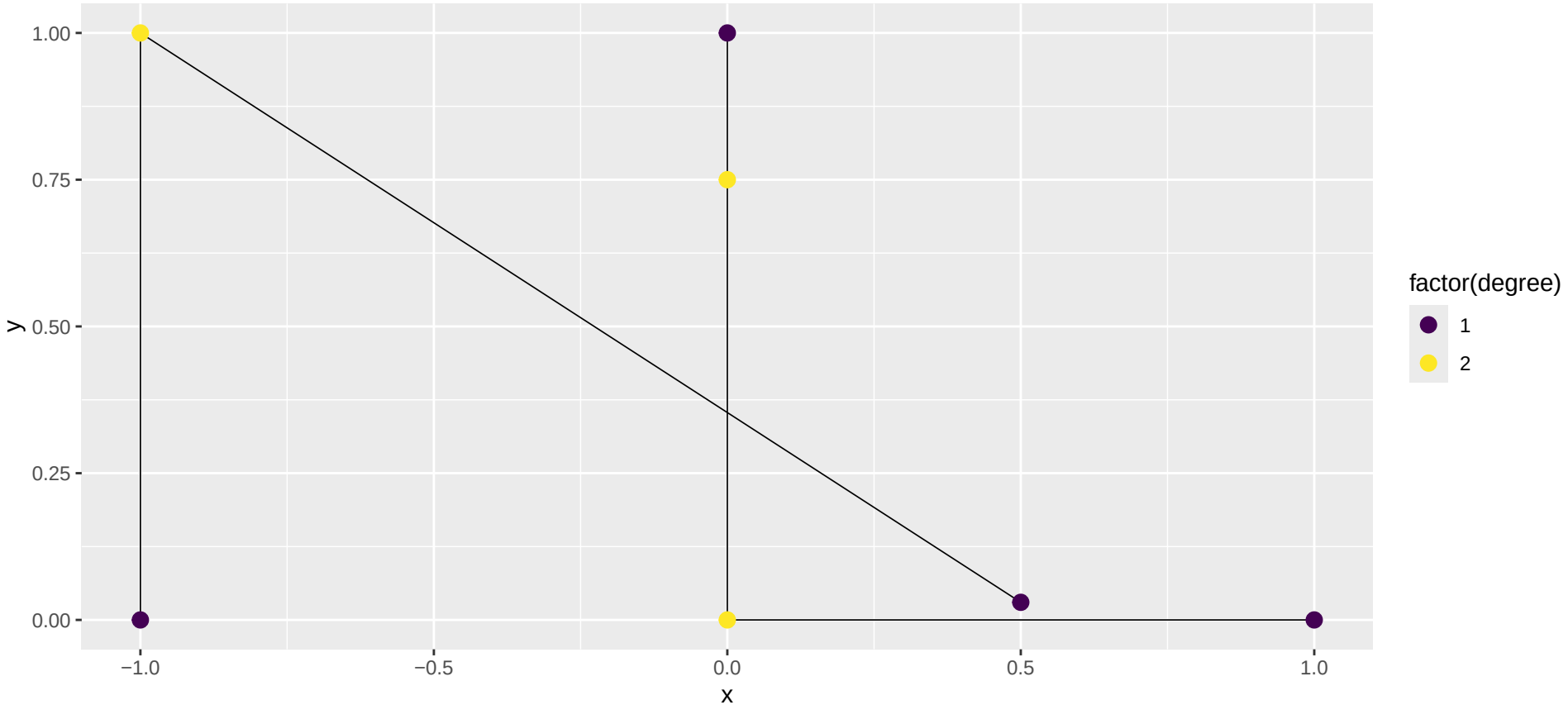
```
graph3$plot(degree = TRUE)
```



Exercise (3)

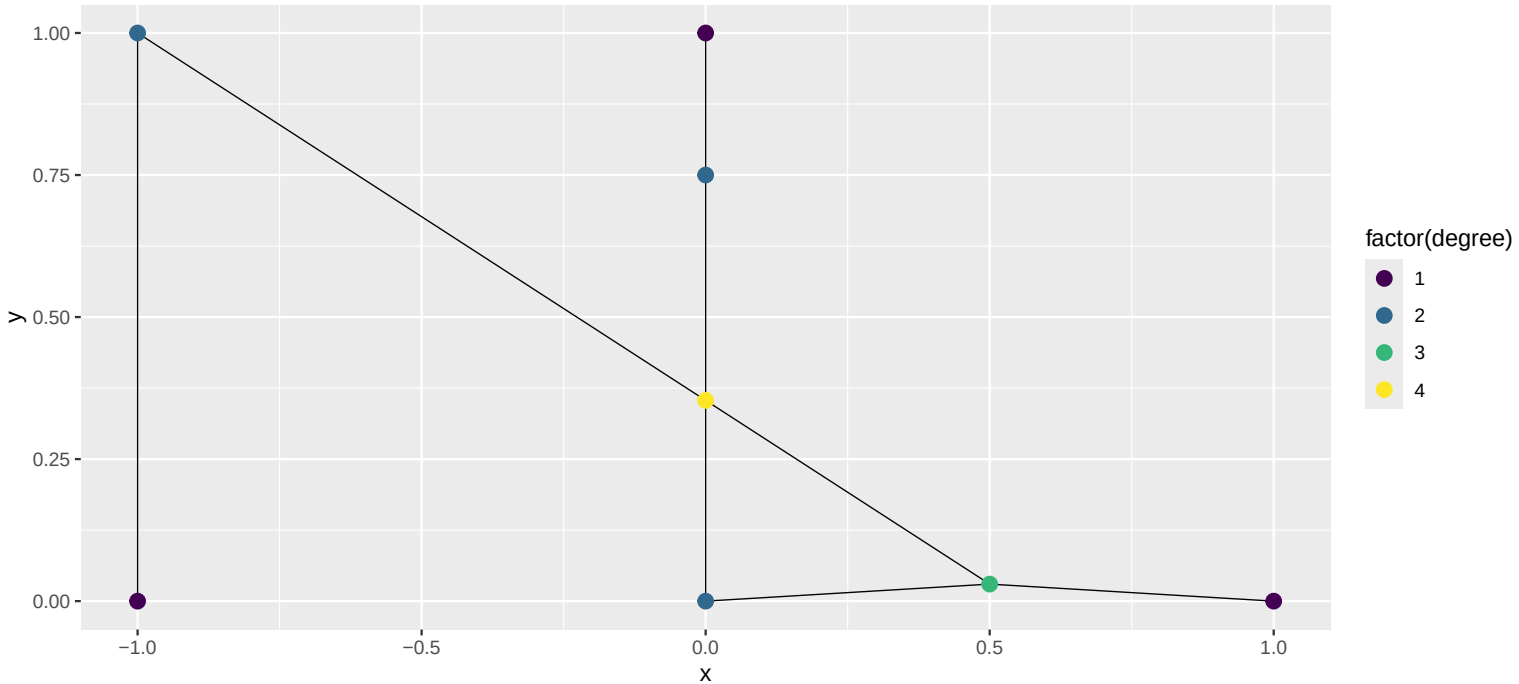
For example, setting the `vertex_vertex` tolerance higher gives:

```
g <- metric_graph$new(edges = edges,
                      perform_merges = TRUE,
                      tolerance = list(vertex_vertex = 0.05))
g$plot(degree = TRUE)
```



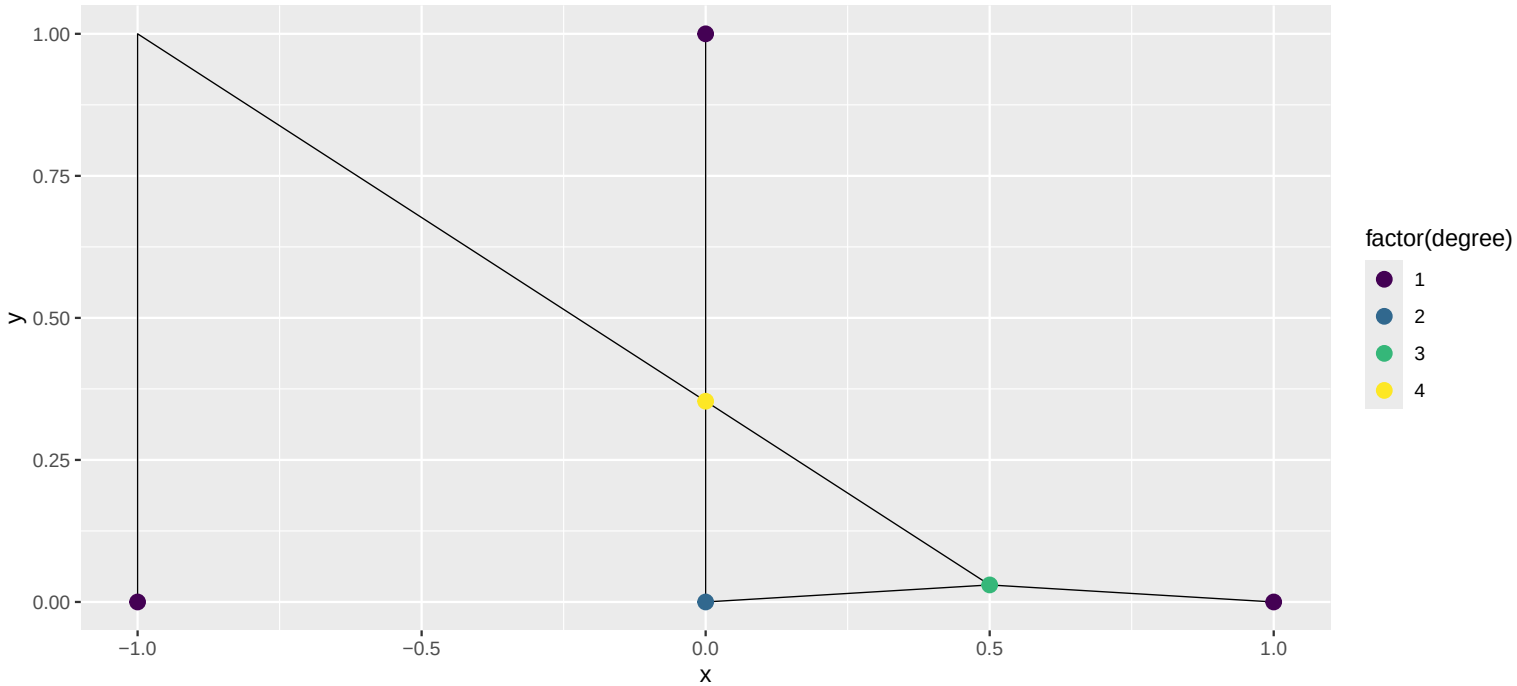
Pruning (1)

```
g1 <- metric_graph$new(edges = edges,
                       perform_merges = TRUE,
                       tolerance = list(vertex_vertex = 0.2,
                                        vertex_edge = 0.1,
                                        edge_edge = 0.001))
g1$plot(degree = TRUE)
```



Pruning (2)

```
g2 <- metric_graph$new(edges = edges,
                       perform_merges = TRUE,
                       tolerance = list(vertex_vertex = 0.2,
                                        vertex_edge = 0.1,
                                        edge_edge = 0.001),
                       remove_deg2 = TRUE)
g2$plot(degree = TRUE)
```

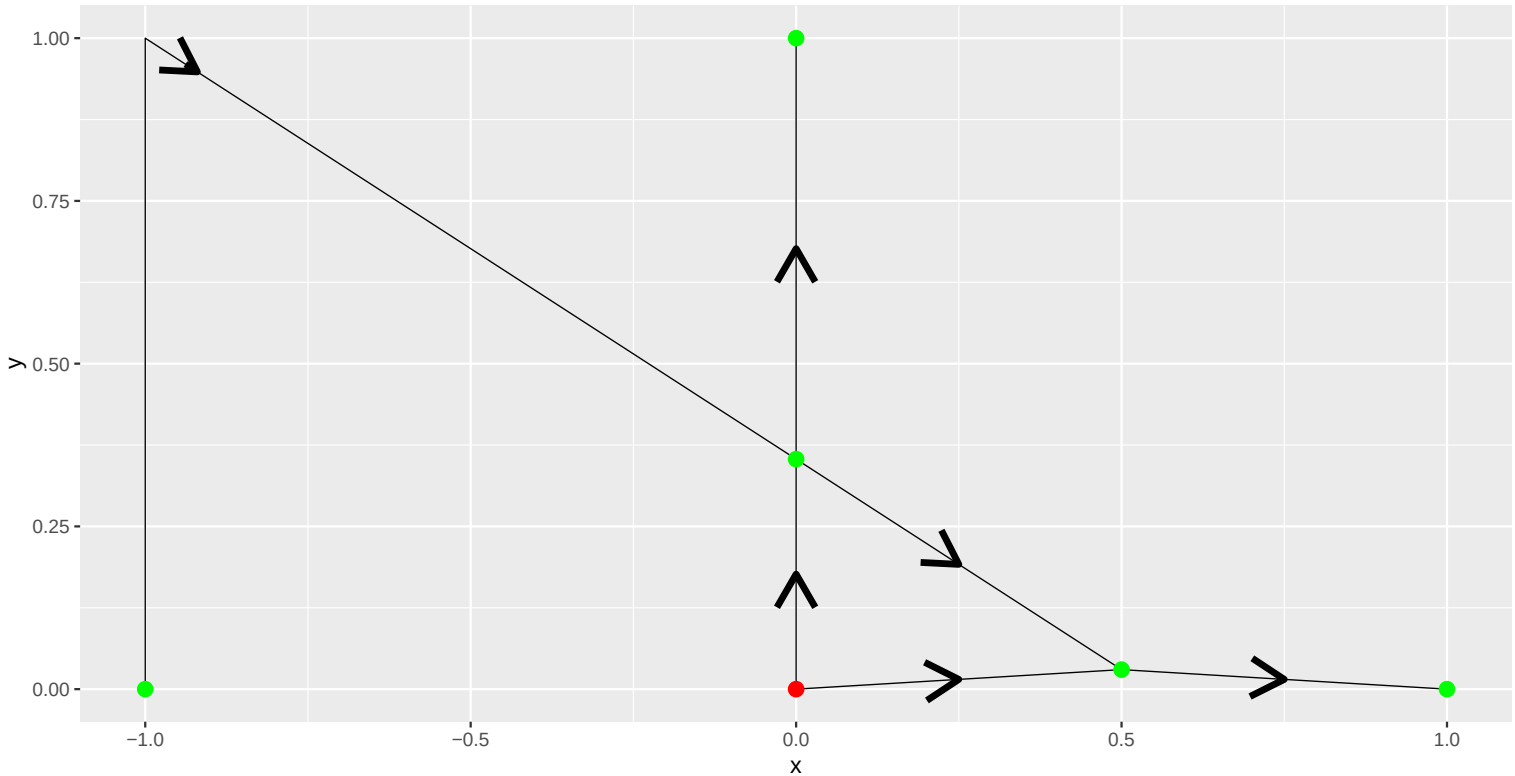


Pruning (3)

Observe that one vertex of degree 2 was not removed.

This is due to the fact that if we consider the direction, this vertex has directions incompatible for removal.

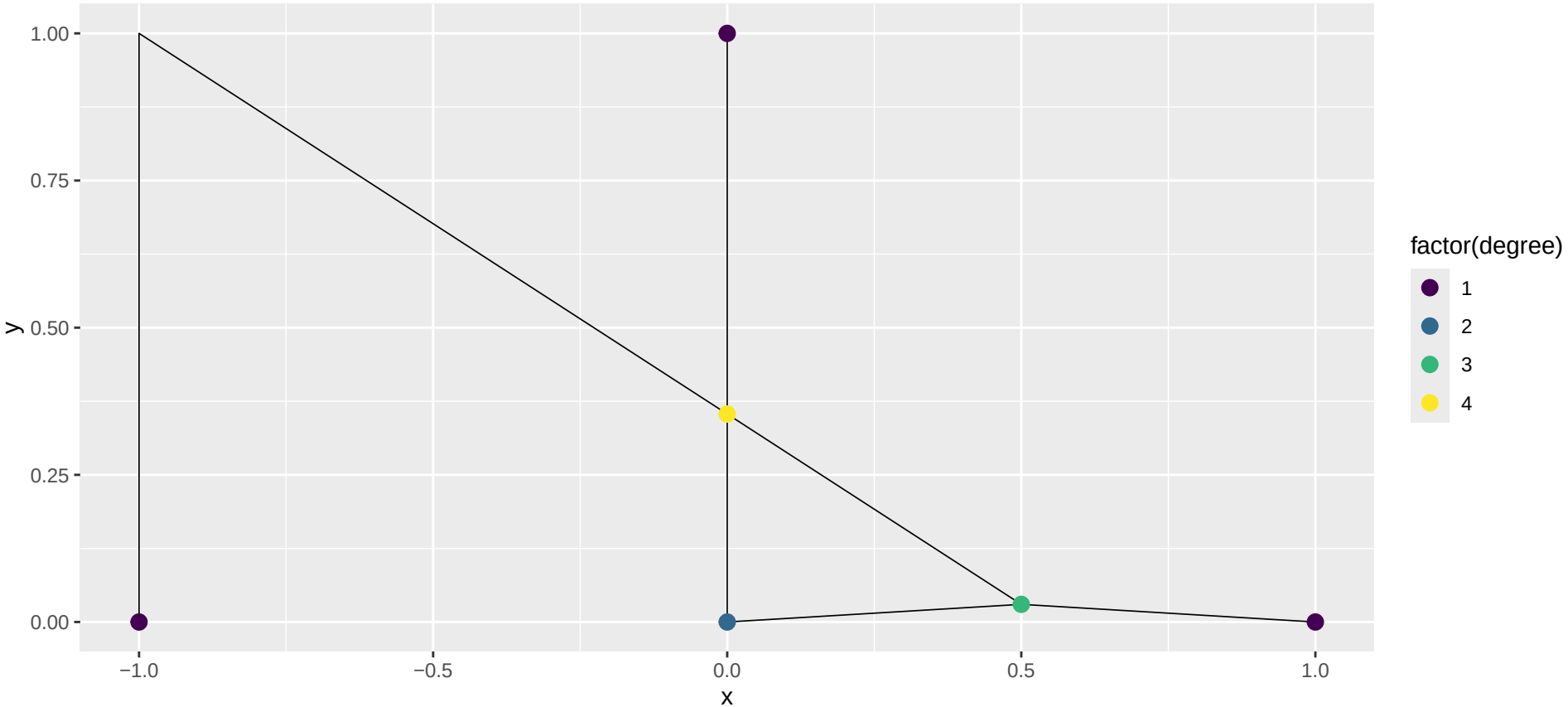
```
g2$plot(direction = TRUE)
```



Pruning (4)

We can also prune the graph after construction:

```
g1$prune_vertices()
g1$plot(degree=TRUE)
```



Properties of graphs

Several informations about the graph are displayed when we print the metric graph object.

Further characteristics of the graph can be obtained by computing the characteristics via the `compute_characteristics()` method.

One may additionally want for more informations (not relevant to the current course) such as distance consistency by using the `check_distance_consistency()` method, and by checking whether it is a graph with Euclidean edges by using the `check_euclidean()` method.

Properties of graphs

```
## A metric graph with  6  vertices and  6  edges.
##
## Vertices:
##   Degree 1: 3;  Degree 2: 1;  Degree 3: 1;  Degree 4: 1;
##   With incompatible directions:  1
##
## Edges:
##   Lengths:
##       Min: 0.3533333  ; Max: 2.190873  ; Total: 4.788107
##   Weights:
##       Min: 1  ; Max: 1
##   That are circles:  0
##
## Graph units:
##   Vertices unit:  None  ; Lengths unit:  None
##
## Longitude and Latitude coordinates:  FALSE
```

Characteristics of the graph

We now compute the characteristics:

```
g1$compute_characteristics()
```

and check them:

```
t(g1$characteristics)
```

```
##       has_loops connected has_multiple_edges is_tree
## [1,] FALSE     TRUE      FALSE               FALSE
```

They are also displayed when one simply prints the metric graph object.

Edges and vertices of a metric graph

Information about the edges can be obtained by looking at the `edges` element of the graph.

Similarly, information about the vertices can be obtained by looking at the `vertices` element of the graph.

One can also look at an specific edge or specific vertex:

```
graph$edges[[4]]
graph$vertices[[2]]
```

Exercise (4)

Consider the graph g1 constructed above before and after pruning. Inspect the graph object, the edge lengths, the edges, vertices, compute their characteristics, and compare them to find differences on a graph when pruning.

```
g1 <- metric_graph$new(edges = edges,
                       perform_merges = TRUE,
                       tolerance = list(vertex_vertex = 0.2,
                                        vertex_edge = 0.1,
                                        edge_edge = 0.001))
```

Coordinates on metric graphs

In a metric graph object we have two types of coordinates:

- Cartesian coordinates: (x, y), where x represents the position in the horizontal axis and y represents the position on the vertical axis;
- Normalized metric graph coordinates (in the package called PtE coordinates): (e, t), where e represents the number of the edge, and t represents the relative position on the edge. The order is the same as the order used to create the metric graph object. So for an edge e connecting v_1 to v_2, we have $(e, 0) = v_1$ and $(e, 1) = v_2$.
- Metric graph coordinates: represented in the same manner as the normalized metric graph coordinates, with the difference that the positions are given in relation to $[0, l_e]$, where l_e is the edge length. Thus, for an edge e connecting v_1 to v_2, we have $(e, 0) = v_1$ and $(e, l_e) = v_2$.

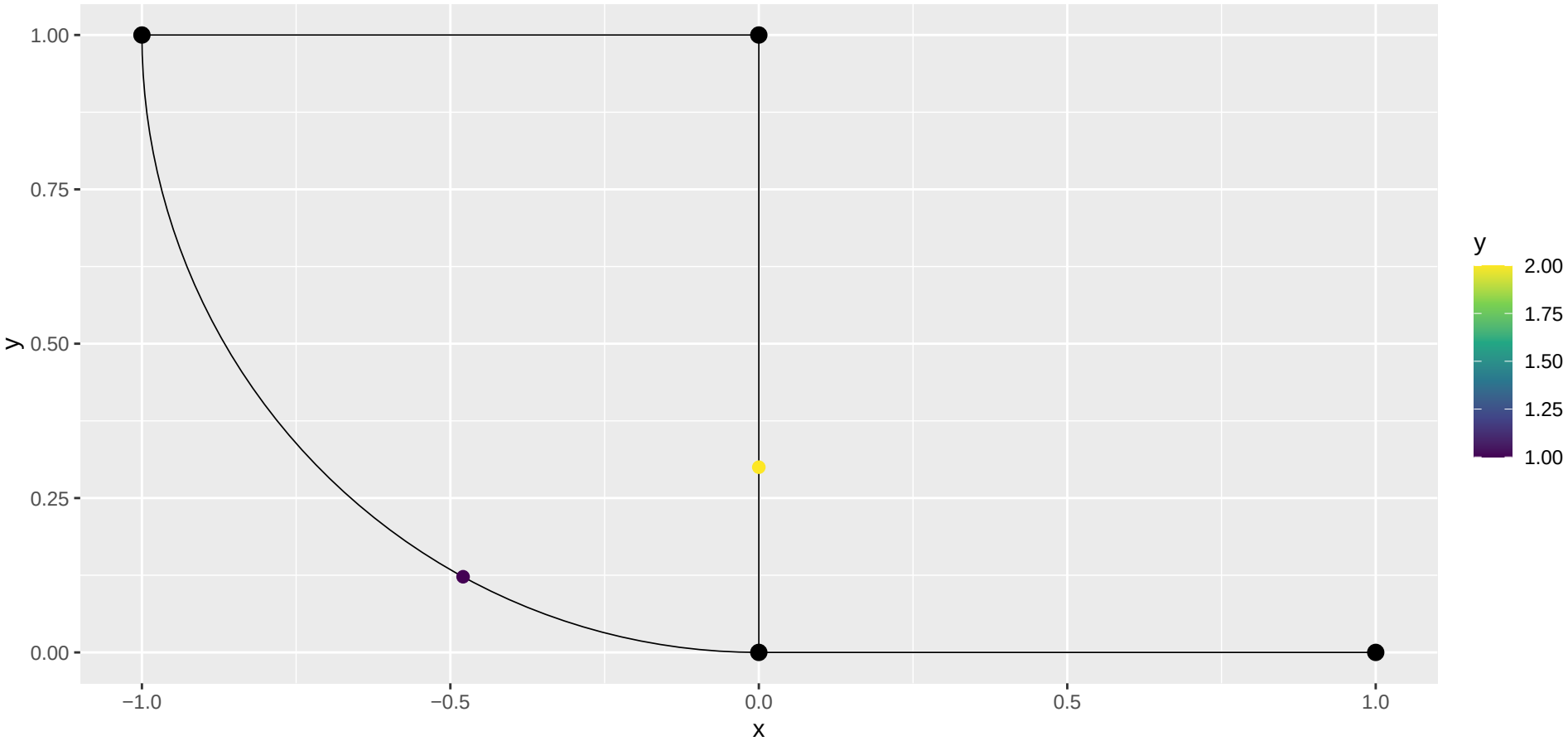
Data on graphs

We can add data into metric graphs by providing a `data.frame` object containing the relevant data and associated coordinates. The associated method is `add_observations()`.

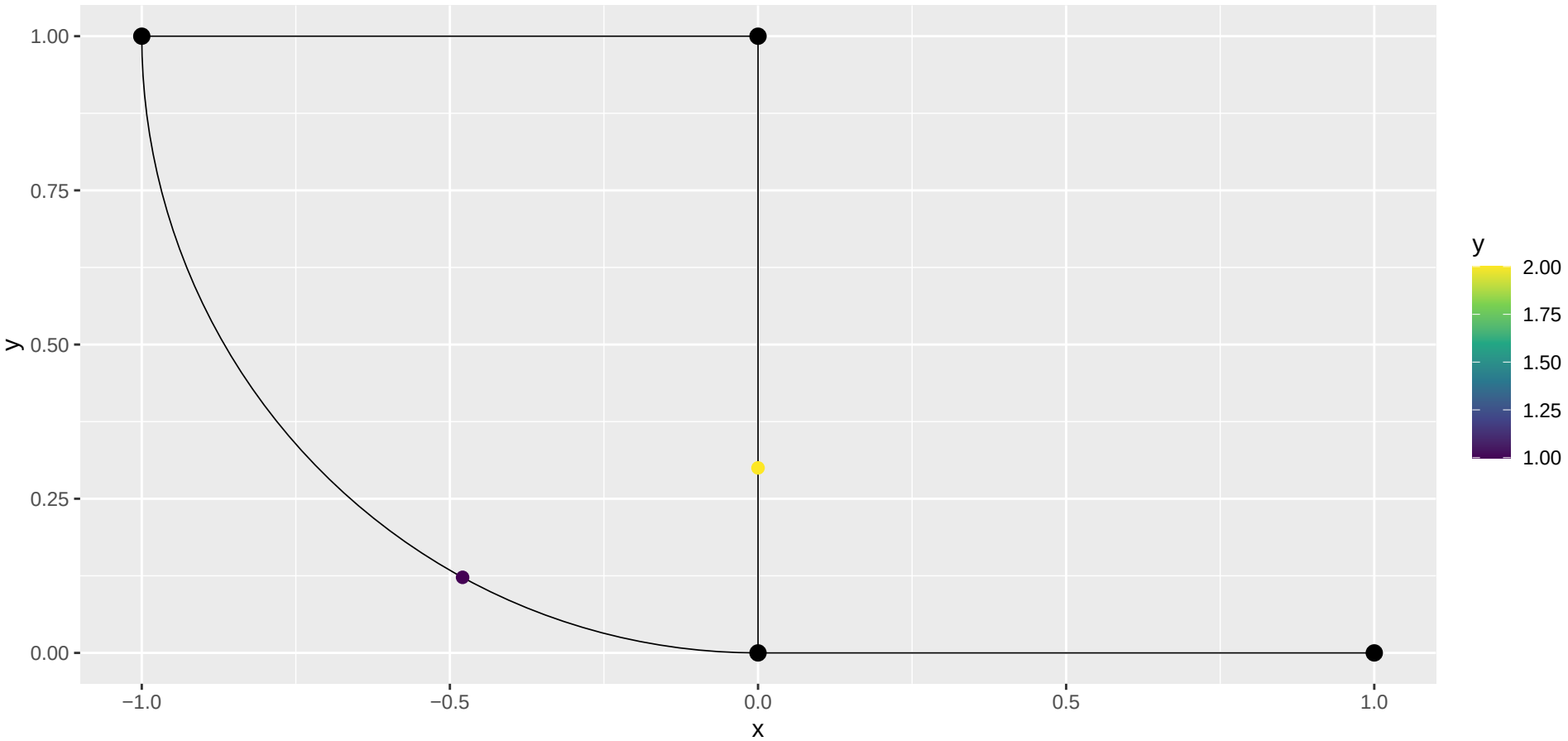
Let us consider a simple example in which we will add data supplying Cartesian coordinates and supplying metric graph coordinates:

```
df_data1 <- data.frame(y = 2, coord_x = 0, coord_y = 0.3)
graph$add_observations(data = df_data1,
                       data_coords = "spatial",
                       coord_x = "coord_x",
                       coord_y = "coord_y")
df_data2 <- data.frame(y = 1, edge_number = 4,
                           distance_on_edge = 0.5)
graph$add_observations(data = df_data2)
```

Data on graph



Data on graph

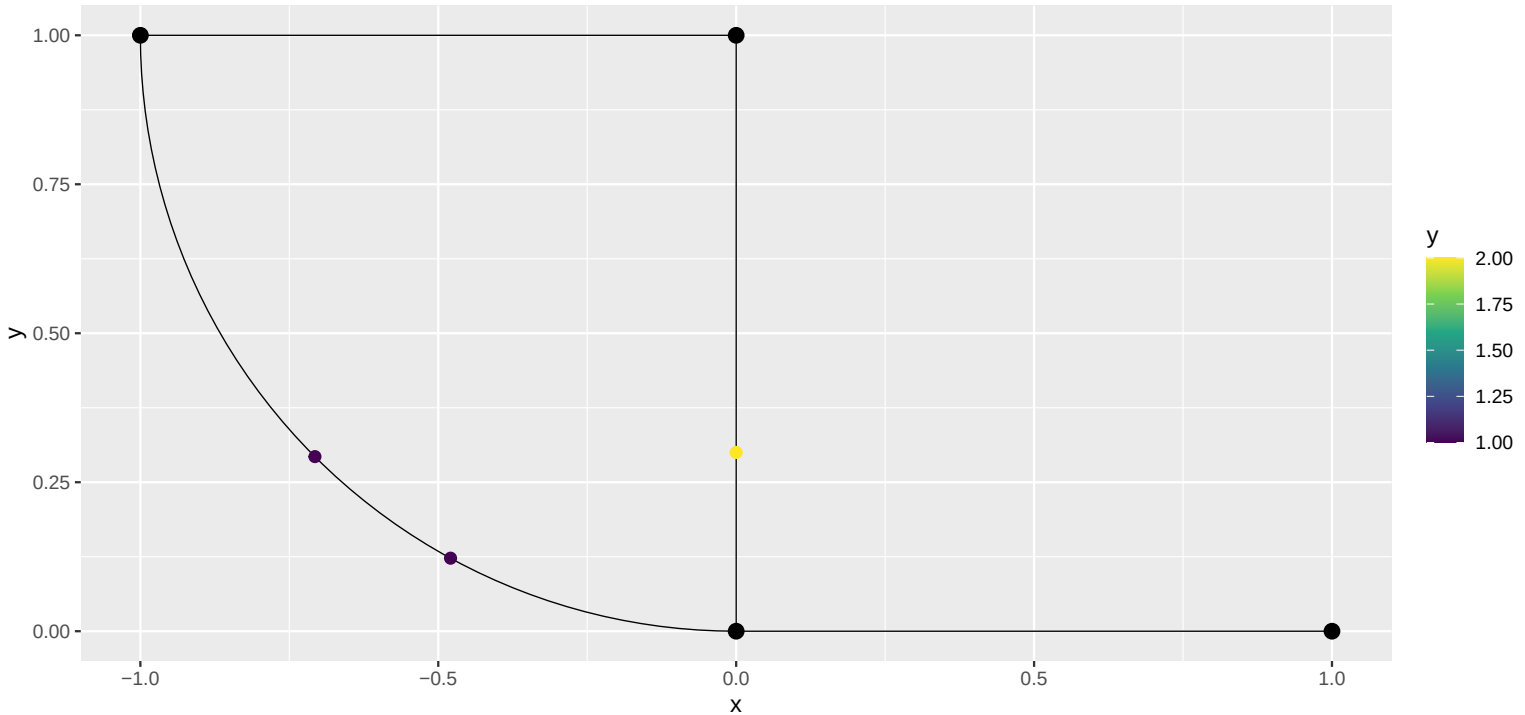


By default, it does not consider normalized coordinates!

Data on graph

To provide data with normalized metric graph coordinates, we must set the argument `normalized` to `TRUE`:

```
graph$add_observations(data = df_data2, normalized = TRUE)
```



Grouped data on graph

We are not allowed to have multiple observations at the same location on a metric graph unless, they are from different groups. More generally, the graph structure also allows to add grouped data. To this end we need to specify which column of the data will be the grouping variable.

To illustrate, let us begin by removing all the observations that are currently in the `graph` object. To this end, we use the `clear_observations()` method:

```
graph$clear_observations()
```

Grouped data on graph

Let us now generate a grouped data:

```
n.repl <- 5
obs_per_edge <- 50
obs_loc <- NULL
for(i in 1:(graph$nE)) {
  obs_loc <- rbind(obs_loc,
                   cbind(rep(i,obs_per_edge),
                   runif(obs_per_edge)))
}

y_repl <- rnorm(n.repl * graph$nE * obs_per_edge)
repl <- rep(1:n.repl, each = graph$nE * obs_per_edge)
```

Grouped data on graph

Now, let us create the `data.frame` with the grouped data, where the grouping variable is `repl`:

```
df_data_repl <- data.frame(y = y_repl, repl = repl,
        edge_number = rep(obs_loc[,1], times = n.repl),
        distance_on_edge = rep(obs_loc[,2], times = n.repl))
```

We can now add this `data.frame` to the graph by using the `add_observations()` method. We need to set `normalized` to `TRUE`, since we have relative distances on edge. We also need to set the `group` argument to `repl`, since `repl` is our grouping variable.

```
graph$add_observations(data = df_data_repl,
                       normalized = TRUE,
                       clear_obs = TRUE,
                       group = "repl")
```

Grouped data on graph

We can check the data inside the metric graph by using the `get_data()` method. Let us check the added data and observe that the grouping variable is now `.group`.

```
head(graph$get_data())[,1:5]
```

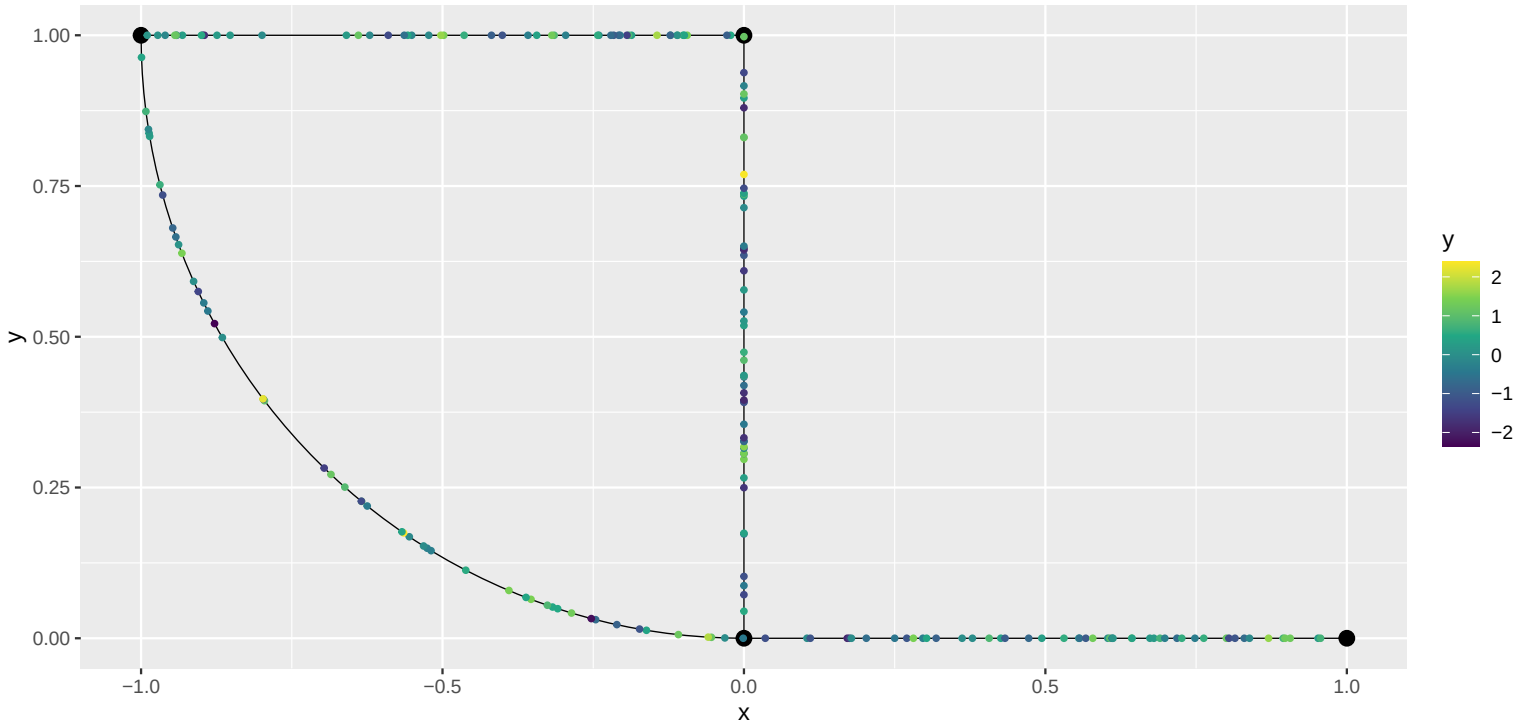
```
## # A tibble: 6 x 5
##         y  repl .edge_number .distance_on_edge .group
##     <dbl> <int>        <dbl>              <dbl> <chr>
## 1 -0.900      1            1             0.0354 1
## 2 -2.18       1            1             0.105  1
## 3 -0.0770     1            1             0.110  1
## 4  0.384      1            1             0.171  1
## 5  1.24       1            1             0.175  1
## 6  1.33       1            1             0.178  1
```

Data on graph - visualization

We can visualize the data on the graph by supplying the `data` argument to the `plot()` method. The `data` argument must be which column of the data inside the metric graph that we want to plot.

When we have multiple groups, we can also choose which group to plot by setting the `group` argument:

```
graph$plot(data = "y", group = 3)
```



Data on graph - Cartesian coordinates

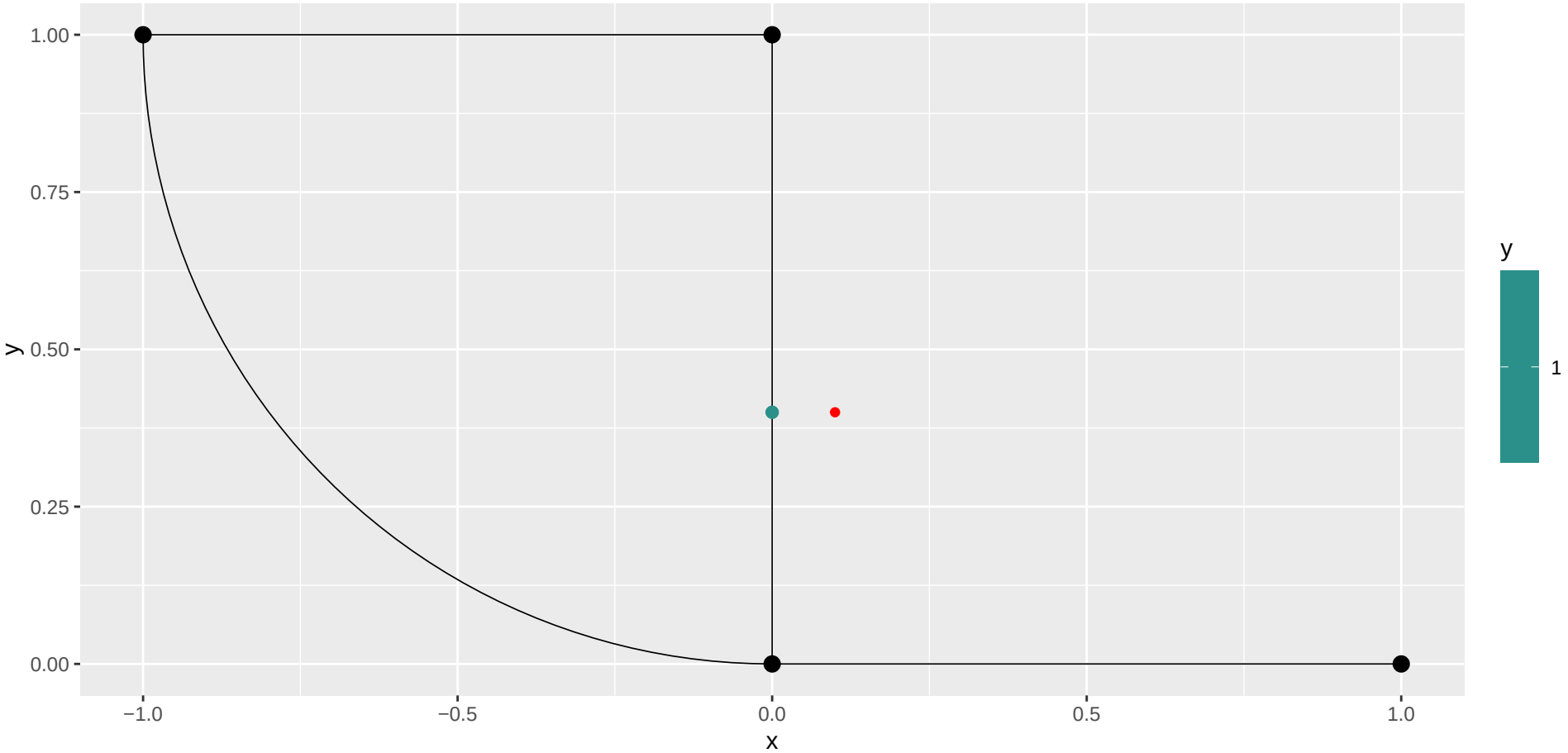
When adding data with Cartesian coordinates, the data will be projected to the closest location on the graph. This is not the case with metric graph coordinates, where the position in the graph is exact.

Let us give an example. We will also clear the current observations by passing the argument `clear_obs` to the `add_observations()` method.

```
df_data3 <- data.frame(y = 1, coord_x = 0.1, coord_y = 0.4)
graph$add_observations(data = df_data3,
                       data_coords = "spatial",
                       coord_x = "coord_x",
                       coord_y = "coord_y",
                       clear_obs = TRUE)
```

Data on graph - Cartesian coordinates

We can see the data was projected to the graph.



Data on graph - Cartesian coordinates

The fact that when dealing with Cartesian coordinates, the data gets projected to the metric graph can sometimes create some issues:

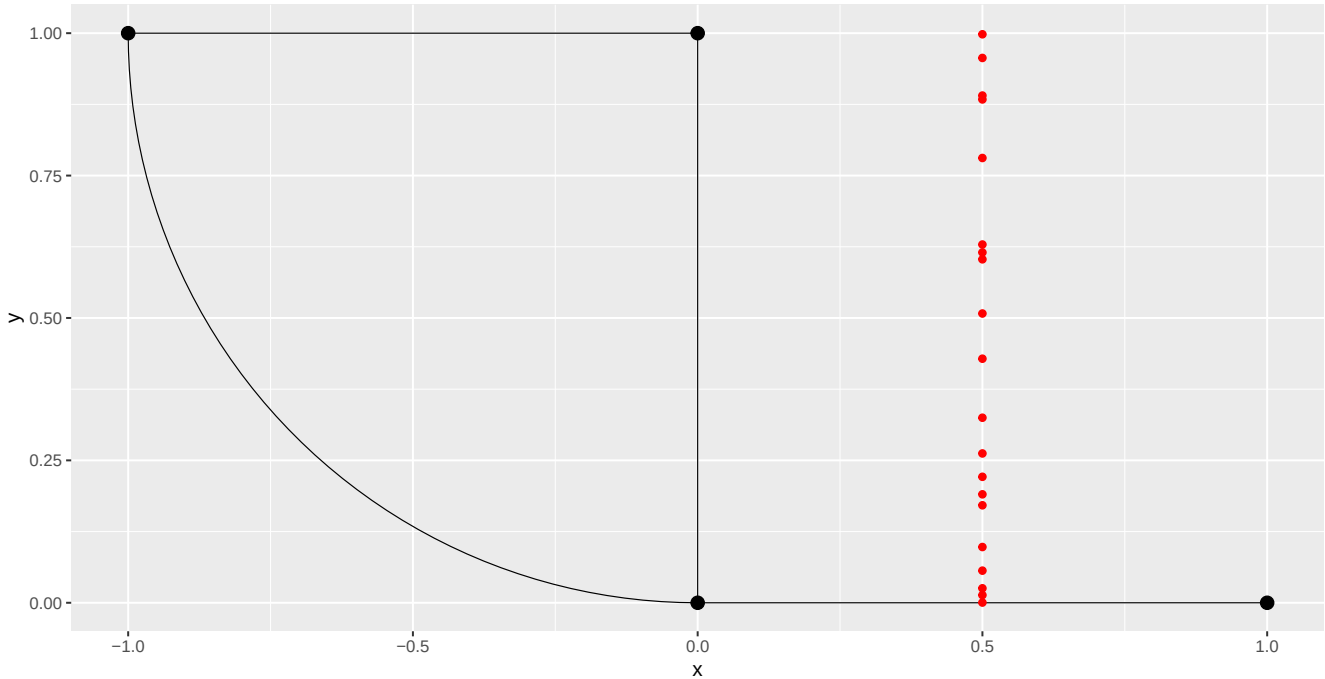
- we can have data from two different spatial locations being projected into the same point;
- we can also have data that is just too far away to make sense to add to the graph.

Data on graph - Cartesian coordinates

Let us illustrate this by including observations that are not contained in the graph.

```
data_df4 <- data.frame(y = rnorm(20),
                       .coord_x = 0.5, .coord_y = runif(20))
```

We start by looking at the graph, and the observations that we created outside the metric graph:



Data on graph - Cartesian coordinates

Let us now add these observations to the graph. The removed observations are returned when calling the `add_observations()` method:

```
rem_obs <- graph$add_observations(data = data_df4,
                                  clear_obs = TRUE,
                                  data_coords = "spatial",
                                  coord_x = ".coord_x",
                                  coord_y = ".coord_y")
```

```
## Adding observations...

## Converting data to PtE

## Warning in system.time({: There were points projected at th
## Only the closest point was kept. To keep all the observatio
## 'duplicated_strategy' to 'jitter'.
```

Observe that we got a warning telling us that we had points projected to the same location.

Data on graph - Cartesian coordinates

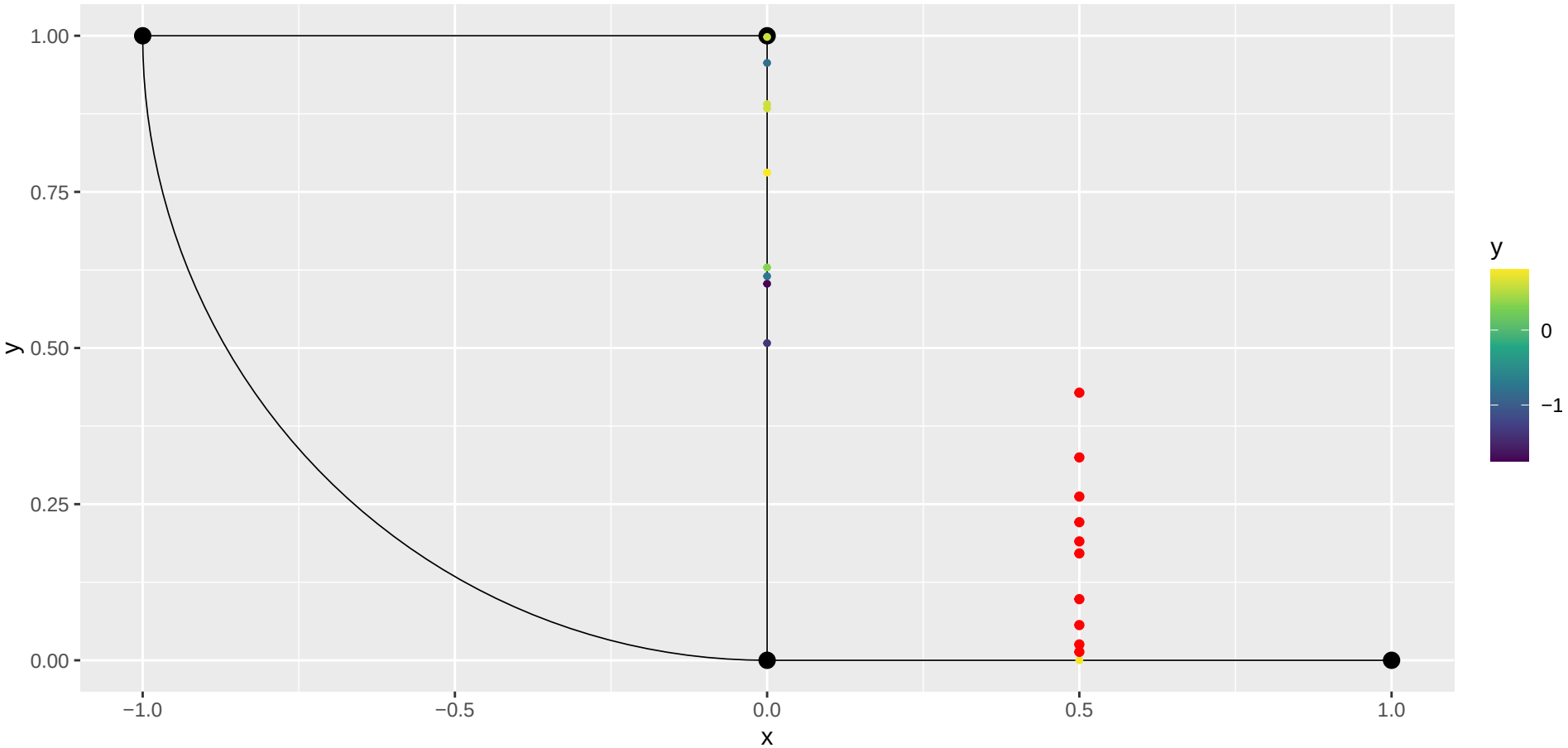
Let us look at the element `rem_obs`:

```
## $removed
##                y .coord_x   .coord_y
## 1  -0.008582728       0.5 0.09786956
## 2  -0.673733897       0.5 0.42849143
## 3  -1.058875141       0.5 0.01347292
## 4  -1.877813593       0.5 0.19037953
## 5  -0.210007629       0.5 0.05640090
## 6   0.492945952       0.5 0.17115332
## 7   0.460878300       0.5 0.32485680
## 8  -0.254317029       0.5 0.02540587
## 9   0.125168825       0.5 0.26215554
## 10  0.302622815       0.5 0.22114057
```

Data on graph - Cartesian coordinates

We can see that there were only data that were removed due to being projected at the same location.

Let us now plot the data, and add the points that were removed in red:



Data on graph - Cartesian coordinates

We can observe that we ended up adding more than one would want due to the tolerance for adding observations in this case.

- Let us reduce the tolerance by using the `tolerance` argument, and let us do a jittering on the observations by setting `duplicated_strategy` argument to `jitter`.
- That is, observations that would be projected at the same place will have a small random displacement to avoid being projected at the same place, and repeat the procedure:

```
rem_obs <- graph$add_observations(data = data_df4,
        clear_obs = TRUE, data_coords = "spatial",
        coord_x = ".coord_x", coord_y = ".coord_y",
        tolerance = 0.1, duplicated_strategy = "jitter")
```

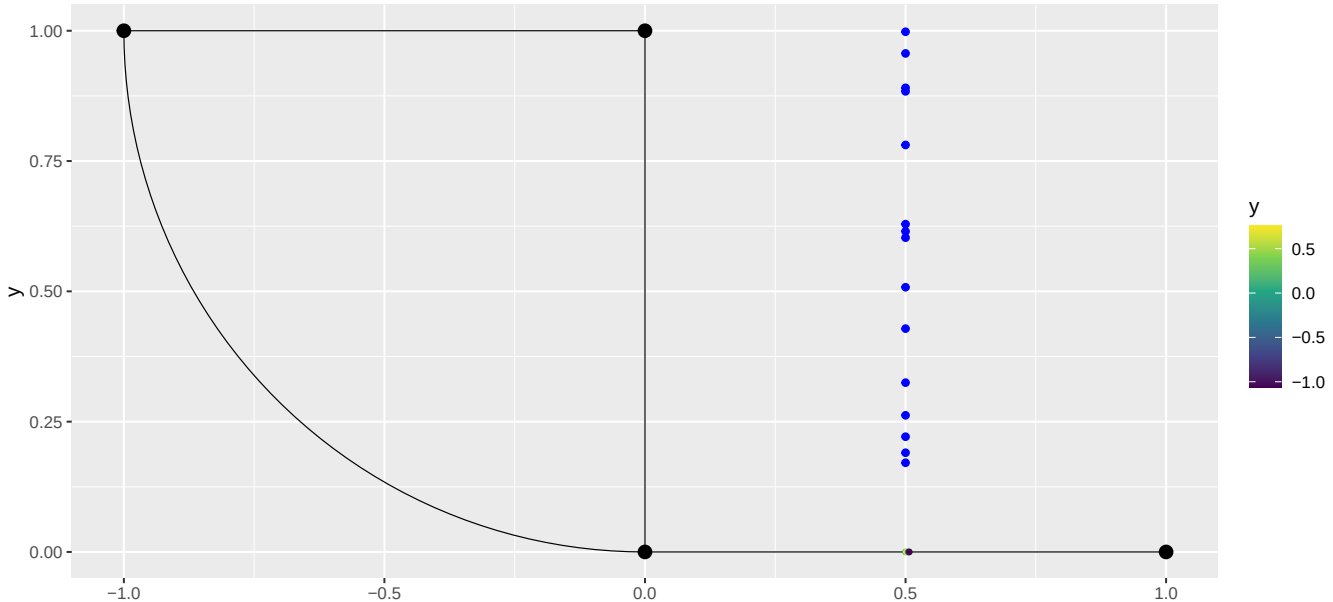
```
## Warning in system.time({: There were points that were farth
## tolerance. These points were removed. If you want them proj
## graph, please increase the tolerance. The total number of p
## being far is 15
```

Data on graph - Cartesian coordinates

Let us now look at `rem_obs`:

```
## List of 1
##  $ far_data:'data.frame':    15 obs. of  3 variables:
##   ..$ y       : num [1:15] 0.613 -0.674 -0.736 -1.75 0.596
##   ..$ .coord_x: num [1:15] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
##   ..$ .coord_y: num [1:15] 0.884 0.428 0.615 0.603 0.89 ...
```

We can see now there were only data removed due to being too far away.
Let us plot the ones that are far away in blue:



Manipulating data

We have implemented tools for manipulating data on metric graphs based on `dplyr::select()`, `dplyr::mutate()`, `dplyr::filter()`, `dplyr::summarise()` and `tidyr::drop_na()`. Observe that these methods do not modify the internal data, it returns the modified data.

We will only illustrate the `mutate()` method, the application of the remaining being similar.

The `mutate` verb creates new columns, or modify existing columns, as functions of the existing columns. Let us create a new column, `new_y`, obtained as the exponential of `y`:

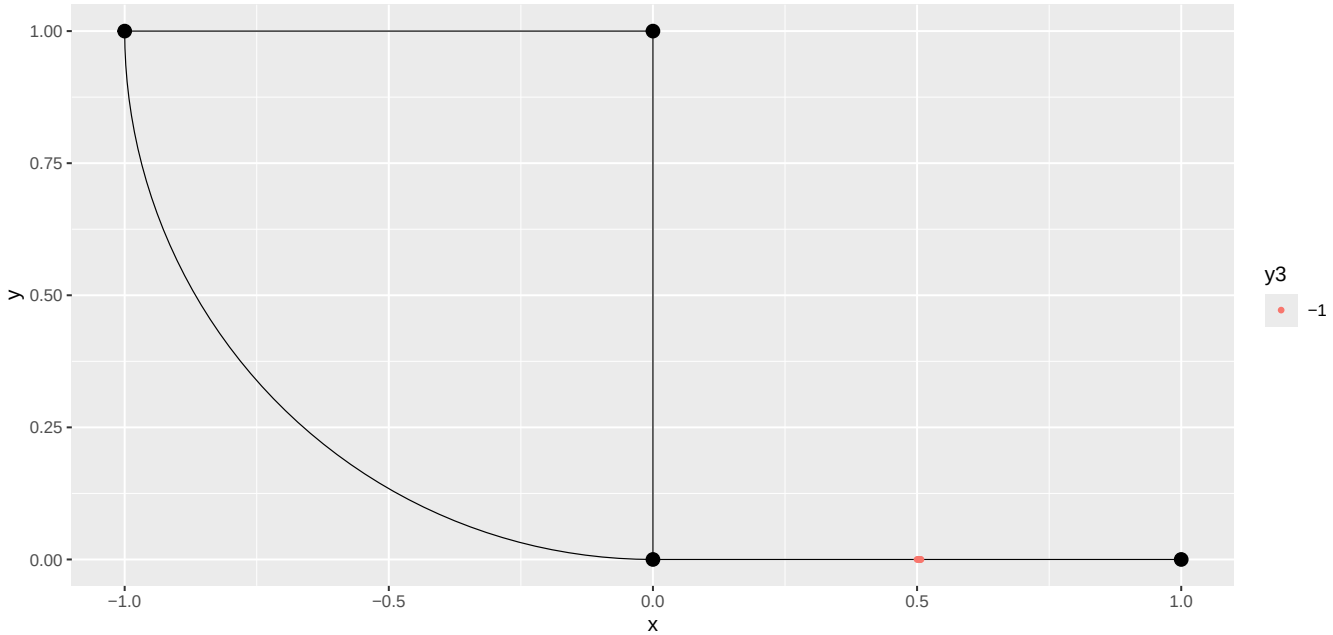
```
graph$mutate(new_y = exp(y))
```

```
## # A tibble: 5 x 8
##          y .distance_to_graph .edge_number .distance_on_edg
##      <dbl>              <dbl>        <dbl>             <dbl
## 1 -0.00858             0.0979            1               0.5
## 2  0.763               0.00178           1               0.50
```

Manipulating data

We can also apply the `mutate()` function the result of the `get_data()` method, and also pipe it to the `plot()` method (we are also changing the scale to discrete):

```
graph$get_data() %>% mutate(new_y = exp(y),
                            y3=as.factor(ifelse(y>1,1,-1))) %>%
                            graph$plot(data = "y3") +
                            scale_colour_discrete()
```



Edge weights

We can add another type of data, which are edge weights. They are data that are constant along the edges.

We begin by creating a dataset of edge weights and adding it to the metric graph:

```
edge_weights_df <- data.frame(
  weight = runif(graph$nE),
  weight2 = rnorm(graph$nE),
  weight3 = runif(graph$nE) * 100
)
```

Edge weights

We can add the edge weights to the graph via the `set_edge_weights()` method:

```
graph$set_edge_weights(weights = edge_weights_df)
```

We can check them inside the graph via the `get_edge_weights()` method:

```
graph$get_edge_weights()
```

```
## # A tibble: 4 x 4
##   weight weight2 weight3 .weights
##    <dbl>   <dbl>   <dbl>    <dbl>
## 1 0.579    0.803    86.1        1
## 2 0.0223   1.73     88.4        1
## 3 0.877   -1.01     98.4        1
## 4 0.851    0.834    66.5        1
```

Manipulating edge weights

Similarly to handling data, we also implemented several methods for edge weights including `select_weights`, `mutate_weights`, `filter_weights`, `summarise_weights`, and `drop_na_weights`. Observe that these methods do not modify the internal edge weights, it returns the modified edge weights.

We will illustrate the `mutate_weights()` method. To this end, let us create a new column `weight_log`, which is the logarithm of `weight + 5`:

```
graph$mutate_weights(weight_log = log(weight + 5))
```

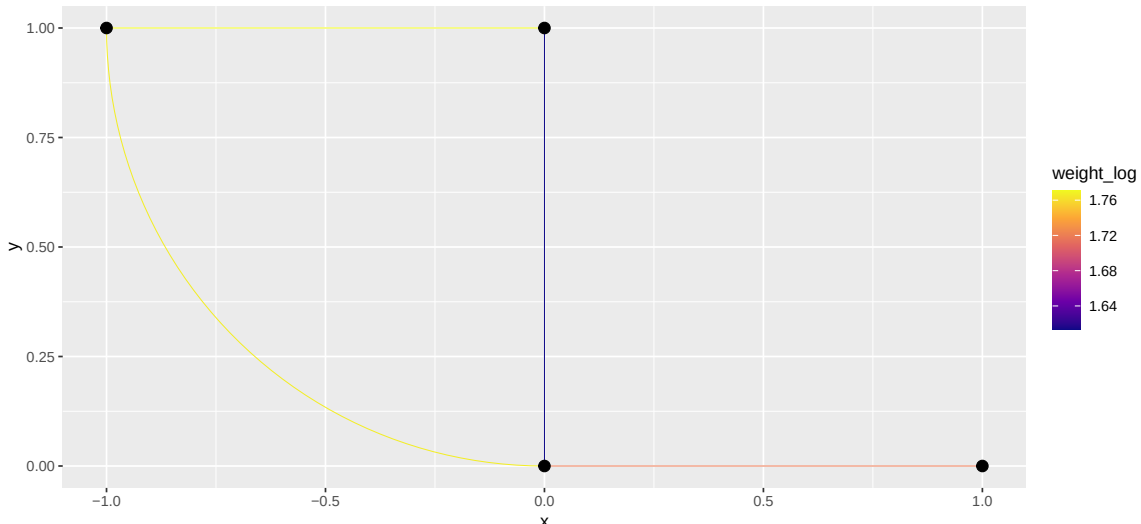
```
## # A tibble: 4 x 5
##   weight weight2 weight3 .weights weight_log
##    <dbl>   <dbl>   <dbl>    <dbl>      <dbl>
## 1 0.579    0.803    86.1        1       1.72
## 2 0.0223   1.73     88.4        1       1.61
## 3 0.877   -1.01     98.4        1       1.77
## 4 0.851    0.834    66.5        1       1.77
```

Edge weights - visualization

We can plot edge weights by using the `plot()` method and passing the column of edge weight we want to plot to the `edge_weight` argument.

It is also noteworthy that we can plot the result after applying the manipulation methods. Let us illustrate with `mutate_weights`. To this end, we first set the new edges as the result from `mutate_weights`, then we plot:

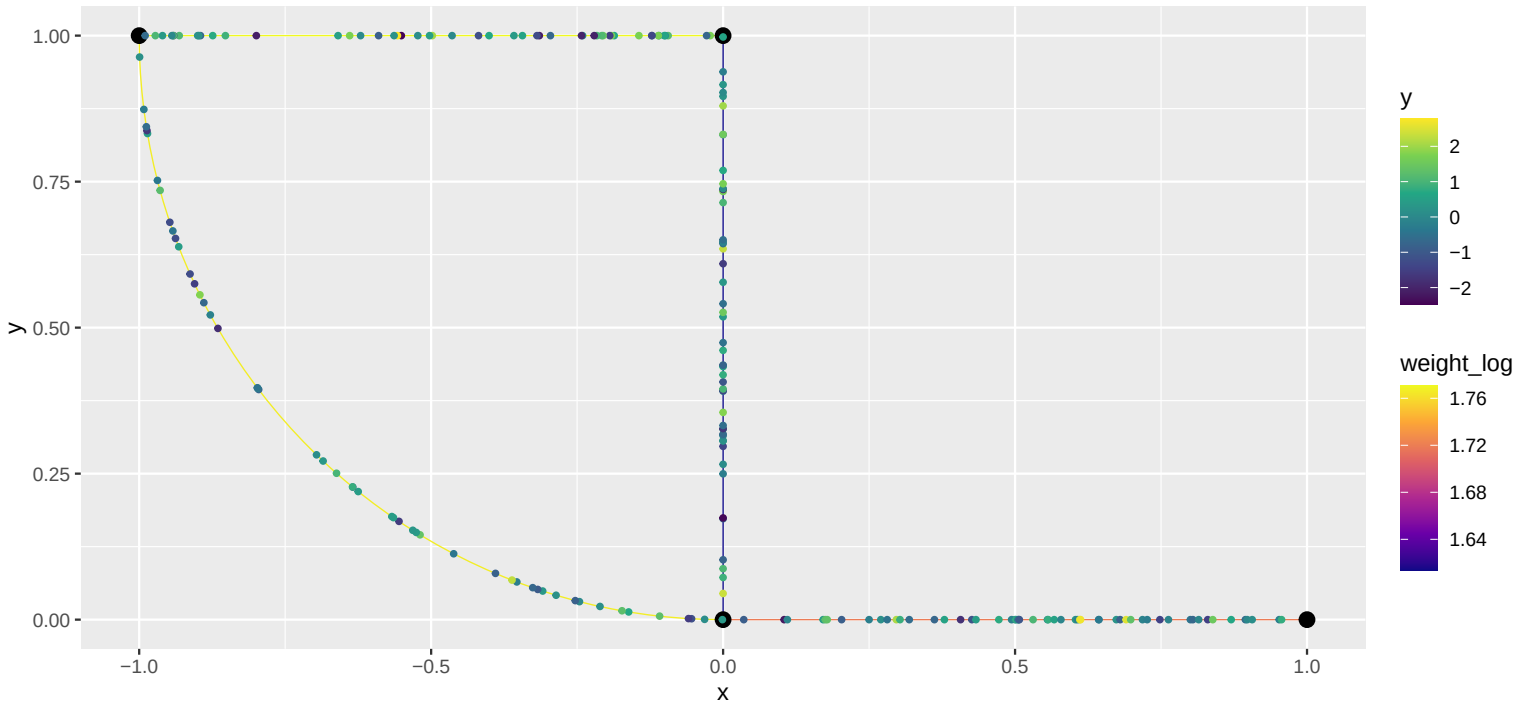
```
graph$set_edge_weights(weights =
    graph$mutate_weights(weight_log = log(weight + 5)))
graph$plot(edge_weight = "weight_log")
```



Data and edge weights - visualization

We can also plot simultaneously the data and edge weights:

```
graph$add_observations(data = df_data_repl, normalized=TRUE,
group = "repl")
graph$plot(data = "y", edge_weight = "weight_log",
           group = 1)
```



Importing graphs

The `MetricGraph` package has interfaces for importing metric graphs from several sources.

Here are some examples:

- Tomtom, overpass turbo, etc;
- `SSN2` package objects;
- `osmdata` package objects;
- `stlnpp` package objects.

Importing graphs - Tomtom, overpass turbo

For Tomtom, overpass turbo, etc, the idea is to convert their exported object into an `sf` object using `st_read()`. Then, we extract the lines and pass them to the `metric_graph` constructor.

For example, consider an object `monaco.geojson` that was exported from overpass turbo.

```
streets <- st_read("monaco.geojson")
```

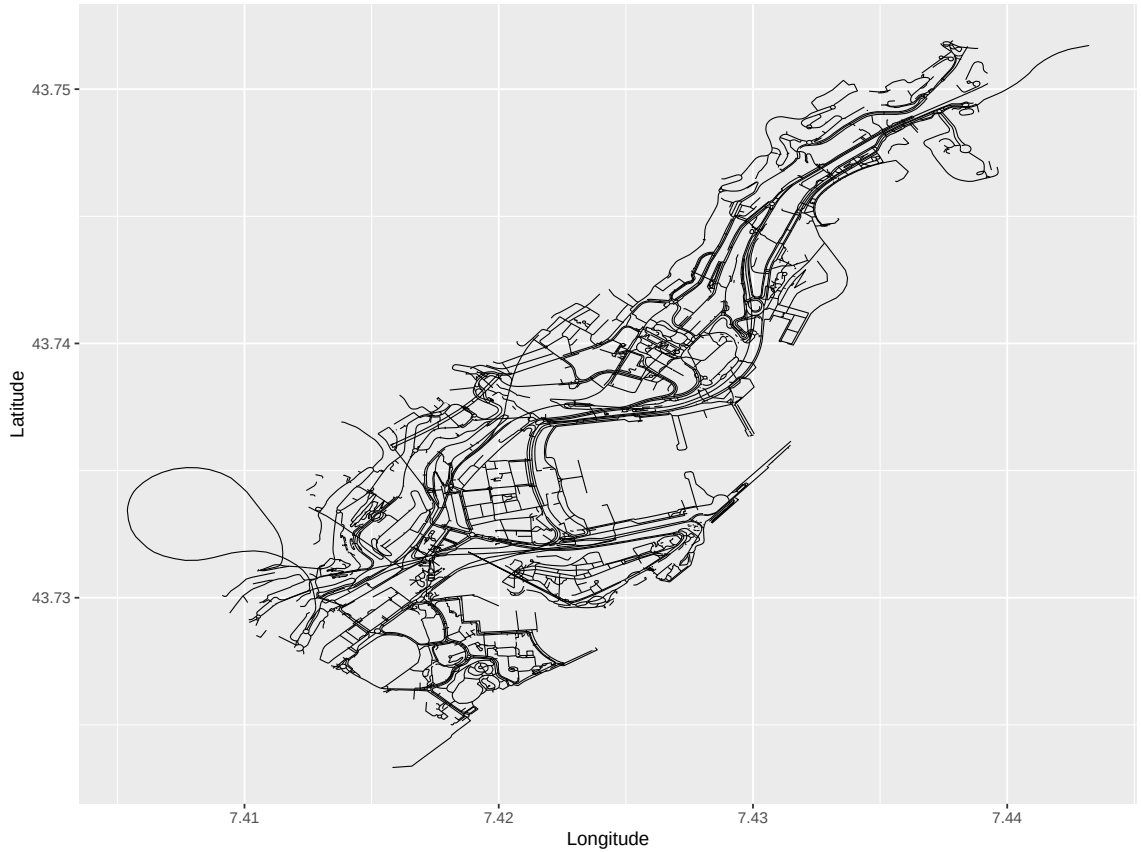
We can then extract the lines and pass them to the `metric_graph` constructor:

```
lines_only <- streets[st_geometry_type(streets) %in%
c("LINESTRING", "MULTILINESTRING"),]
```

- When importing `Tomtom`, `Overpass Turbo`, edge weights are automatically imported.

Importing graphs - Tomtom, overpass turbo

```
monaco_graph <- metric_graph$new(lines_only)
monaco_graph$plot(vertex_size = 0)
```



Importing graphs - SSN2 package

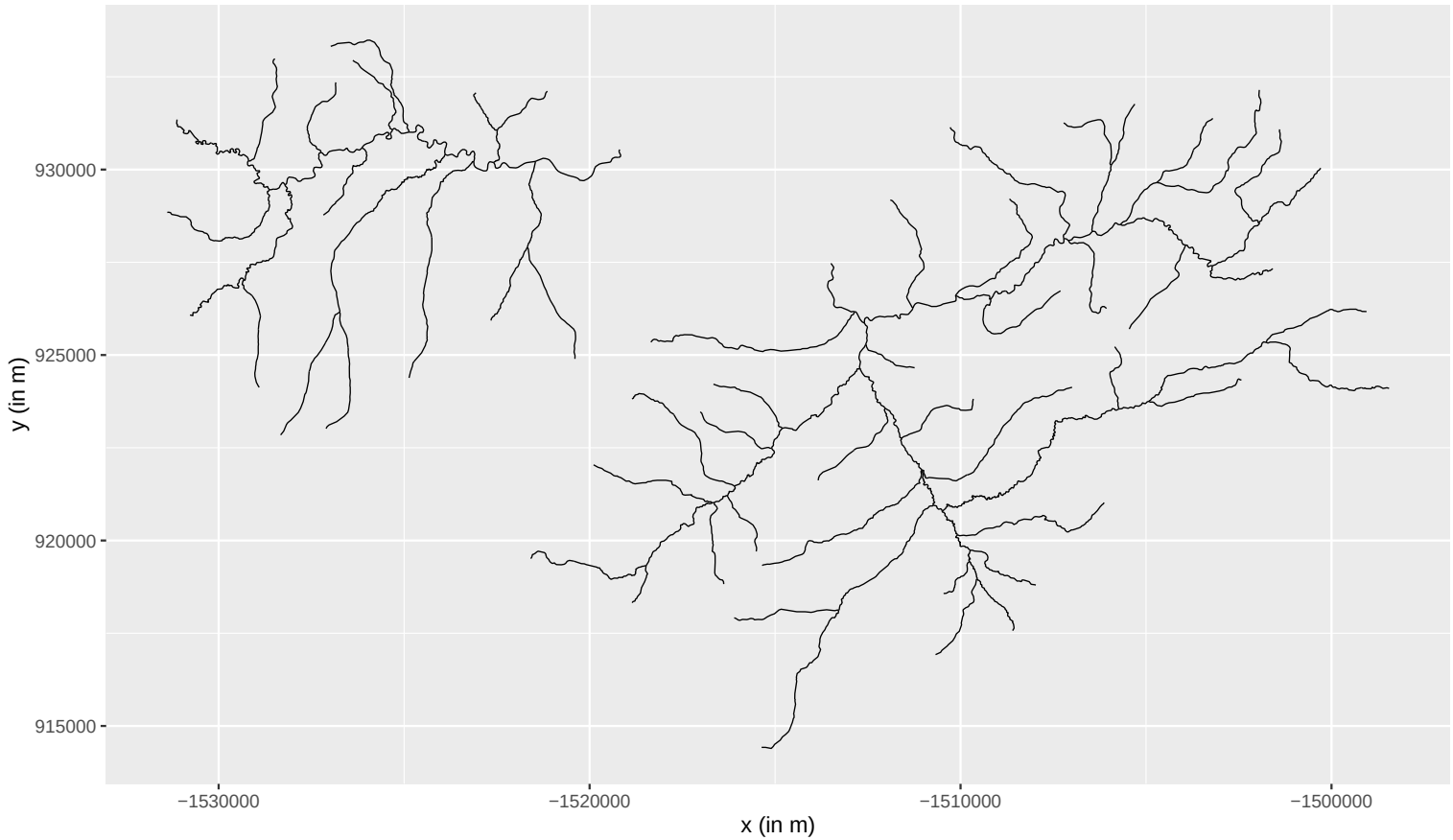
We can also import SSN2 package objects. To build the graph, all we need to do is to pass the SSN2 object to the metric_graph constructor.

```
library(SSN2)
copy_lsn_to_temp()
path <- file.path(tempdir(), "MiddleFork04.ssn")
SSN2_object <- ssn_import(path, overwrite = TRUE)
ssn2_graph <- MetricGraph::metric_graph$new(SSN2_object)
```

- When importing SSN2 objects, both data and edge weights are automatically imported.

Importing graphs - SSN2 package

```
ssn2_graph$plot(vertex_size = 0)
```



Importing graphs - osmdata package

The osmdata package can be used to import OpenStreetMap data. To build the graph, we need to pass the osmdata object to the metric_graph constructor.

```
library(osmdata)
call <- opq(bbox = c(39.0884, 22.33, 39.115, 22.3056))
call <- add_osm_feature(call, key = "highway",
                        value=c("motorway",
                        "primary","secondary",
                        "tertiary",
                        "residential"))
osm_data <- osmdata_sf(call)

osm_graph <- metric_graph$new(osm_data)
```

- When importing osmdata objects, edge weights are automatically imported.

Importing graphs - osmdata package

```
osm_graph$plot(vertex_size = 0)
```

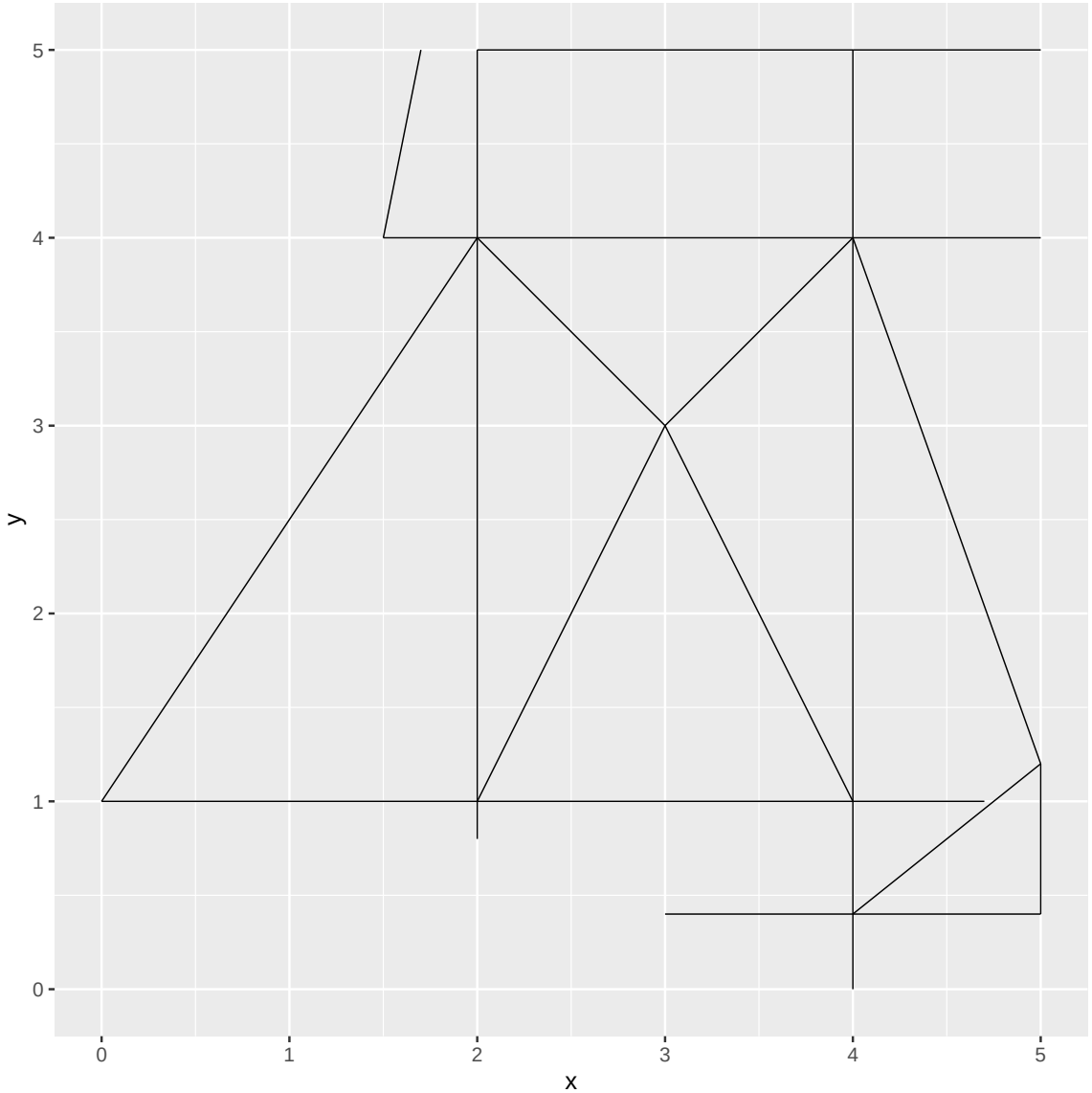
Importing graphs - stlnpp package

We can also import stlpp objects from the stlnpp package. To this end, we need to use the stlpp.to.graph function and we need to supply the coordinate reference system.

```
library(stlnpp)
data(easynet)
X <- rpoislpp(1,easynet)
t <- runif(npoints(X))
stlpp <- stlpp(X,T=t,L=easynet)
stlpp_graph <- stlpp.to.graph(stlpp, crs = st_crs(NA))
```

Importing graphs - stlnpp package

```
stlpp_graph$plot(vertex_size = 0)
```



Handling CRS

When creating metric graphs based on real world data, it is very important to handle the coordinate reference system (CRS). The main reason is that this will ensure that the computed quantities, such as edge lengths, are correct.

- When creating a metric graph from an `sf` object, if the object has a CRS, then the CRS is automatically set in the metric graph creation.
- When importing data from `osmdata` and `SSN2` packages, the CRS is automatically set, as it is part of the object.
- When importing data from `stlnpp` package, we need to explicitly set the CRS.
- When importing data from `Tomtom`, `Overpass Turbo`, etc, the object typically has CRS.
- The easiest way to handle the CRS is to set it on the `sf` object before importing it to the metric graph.

Handling CRS - Adding data with CRS

When we add data with Cartesian coordinates to a metric graph with a CRS, it is important to make sure that the data is an `sf` or `sp` object and that it also has a CRS.

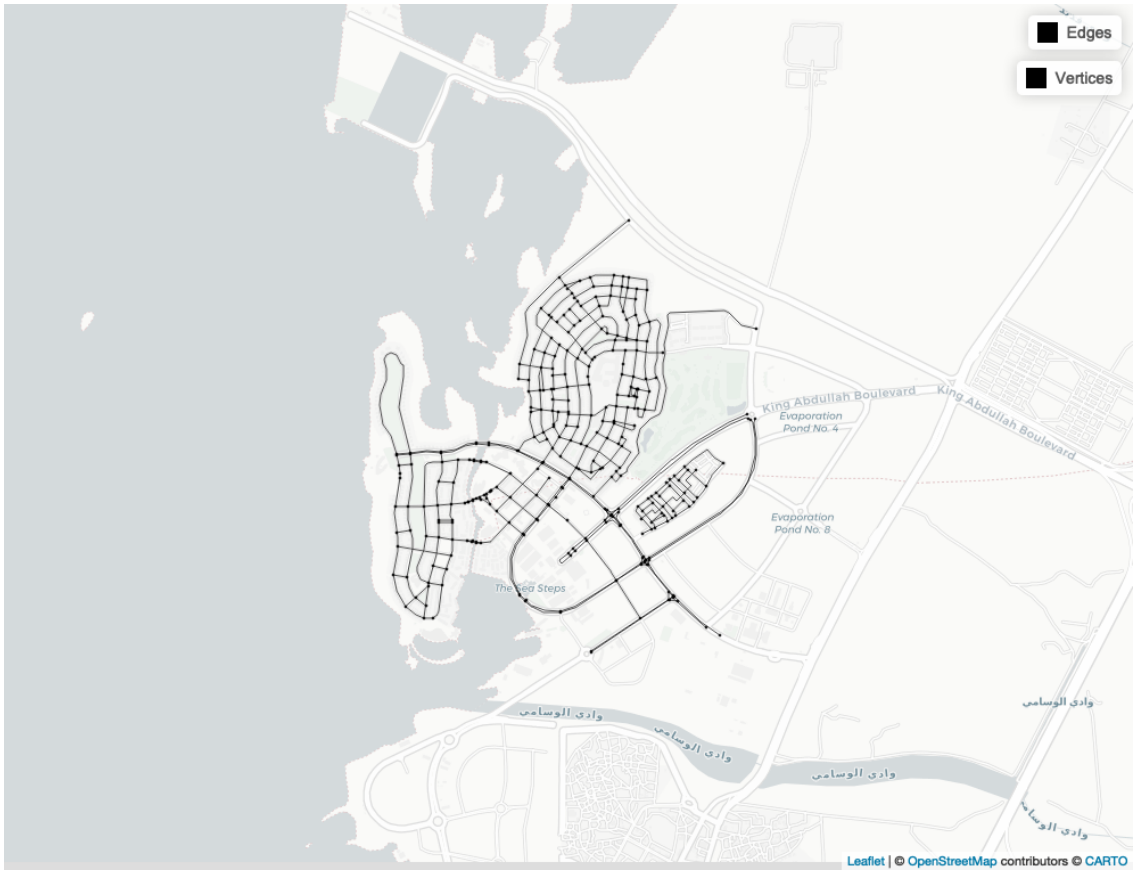
- If the data does not have a CRS, the CRS from the metric graph is used.
- The CRS from the data does not need to match the CRS of the metric graph.

Therefore, when handling metric graphs from the real world, it is important to make sure that the both the metric graph and data have CRS.

Plotting with mapview

We have an interface to the `mapview` package, which creates an interactive map. To use it, we simply set the `type` argument to `mapview` in the `plot()` method.

```
osm_graph$plot(vertex_size = 0.5, type = "mapview")
```



Graph components (1)

Observe that the constructor showed the message that the graph is not connected. This might not be ideal for modeling and we may want to study the different connected components separately. If this is not a concern, one can set the argument `check_connected = FALSE` while creating the graph. In this case the check is not done and no warning message is printed.

Graph components (2)

- To construct all connected components, we can create a `graph_components` object:

The `graph_components` class contains a list of `metric_graph` objects, one for each connected component. In this case, we have

```
## [1] 45
```

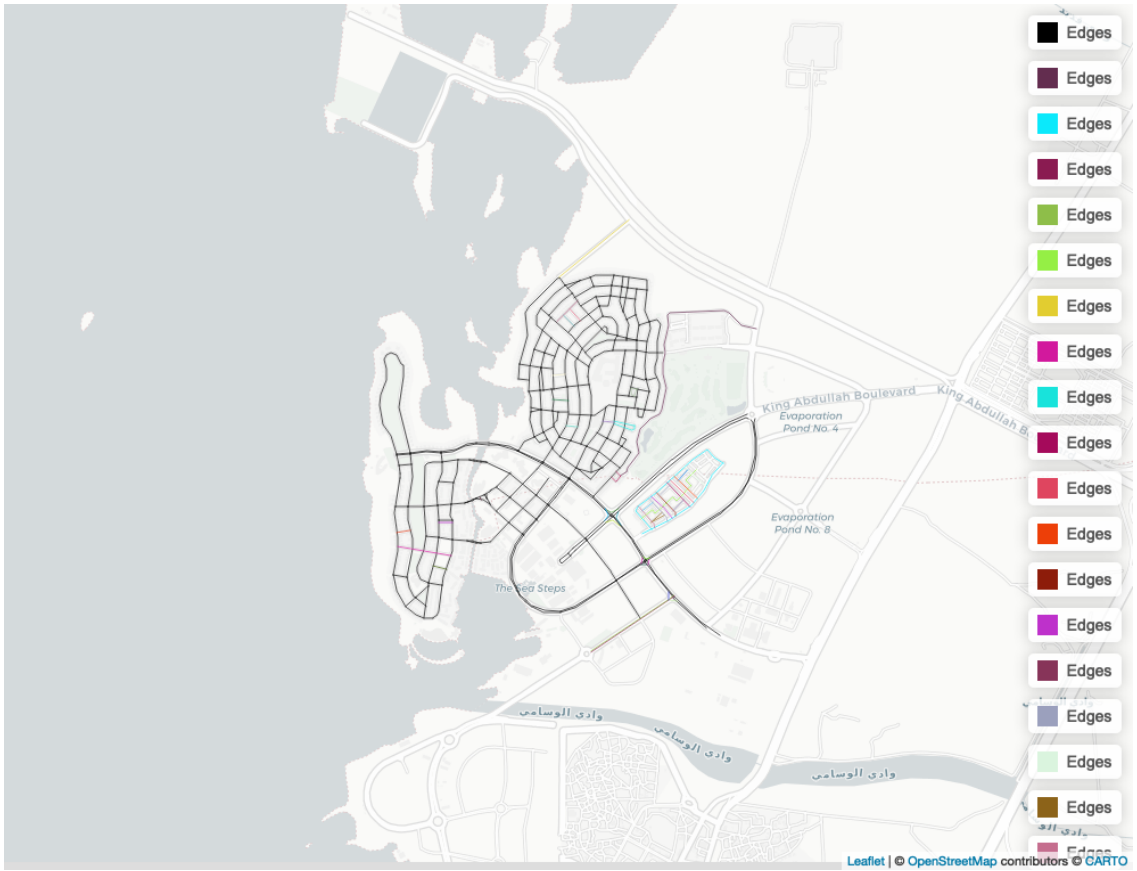
components in total, and their total edge lengths in km at

```
## Units: [km]
##  [1] 57.17622451  2.14141501  1.93532168  0.77465701  0.773
##  [7]  0.69818010  0.41667501  0.36334762  0.26076350  0.254
## [13]  0.23636407  0.23040866  0.22719590  0.22616730  0.223
## [19]  0.20803216  0.19722992  0.14829543  0.12946764  0.117
## [25]  0.11688631  0.11275246  0.11269753  0.11008294  0.110
## [31]  0.10682621  0.10591008  0.10223391  0.10051815  0.099
## [37]  0.09703292  0.08606433  0.06593386  0.06174274  0.048
## [43]  0.04322574  0.04147878  0.01783664
```

Graph components (3)

To plot all of them, we can use the `plot` command of the class:

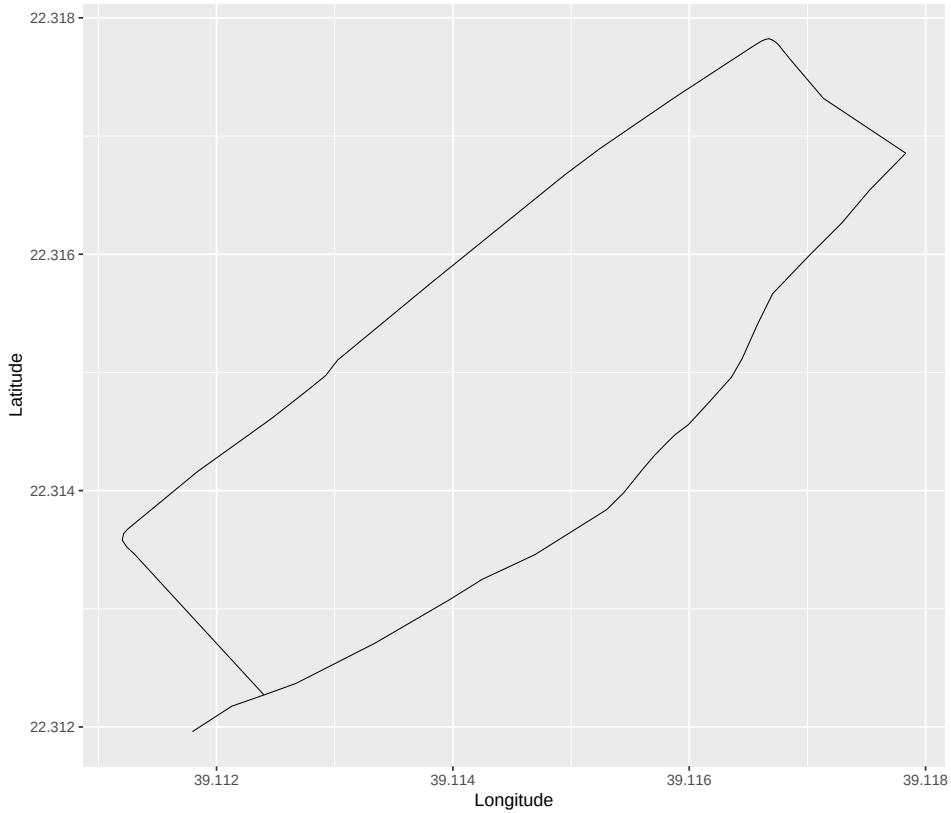
```
osm_graphs$plot(vertex_size = 0, type = "mapview")
```



Graph components (4)

We can retrieve each connected component as a standard `metric_graph` object to work with. For example, we can retrieve the third connected component:

```
osm_graph3 <- osm_graphs$graphs[[3]]
osm_graph3$plot(vertex_size = 0)
```



Graph components (5)

We can also retrieve the largest connected component via the `get_largest` command:

```
osm_graph <- osm_graphs$get_largest()
osm_graph$plot(vertex_size = 0)
```