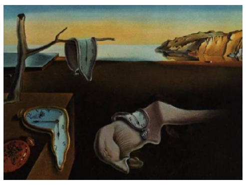


Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

cubble: An R Package for Structuring Spatio-temporal Data

H. Sherry Zhang
Monash University

Dianne Cook
Monash University

Ursula Laa
University of Natural Resources and Life Sciences

Nicolas Langrené
CSIRO Data61

Patricia Menéndez
Monash University

Abstract

The abstract of the article.

Keywords: spatio-temporal data, R.

1. Introduction

Motivation

Many data structures have been proposed for spatial (`sf` by Pebesma (2018)) and temporal (`tsibble` by Wang, Cook, and Hyndman (2020)) data in the R community, while less has been done for spatio-temporal data. The lack of such tools could potentially because analysts usually treat the spatial and temporal dimension pf the data separately, without realising the need to create a new data structure. While this approach follows the third tidy data principal (Wickham 2014) (*Each type of observational unit forms a table*), analysts always need to manually join results from different observational units or combining multiple tables into one for downstream analysis. This additional step doesn't add new operations into the data but can be error prone.

Existing packages

Currently, available spatio-temporal data structure in R includes: `spacetime`(Pebesma 2012), which proposes four space-time layouts: Full grid (STF), sparse grid(STS), irregular (STI), and trajectory (STT). The data structure it uses is based on `sp` (Pebesma and Bivand 2005) and `xts`(Ryan and Ulrich 2020), both of which has been replaced by more recent implementations. `spatstat` (Baddeley and Turner 2005) implements a `ppp` class for point pattern data; and more recent, `stars` (Pebesma 2021) implements a spatio-temporal array with the dplyr's data cube structure `cubelyr` (Wickham 2020) as its backend. While these implementations either store spatial and temporal variables all in a single table, hence duplicate the spatial variables for each temporal unit; or split them into two separate tables that has the problem of manually joining, mentioned in the previously. None of these packages enjoy both the benefits of being able to separate manipulation in the two dimensions while also keep the data object as a whole. This create a gap in the software development. The requirement for such a tool is important given the ubiquity of spatio-temporal vector data in the wild: the Ireland wind data from `gstat` is an classic example data that splits variables into spatial (`wind.loc`) and temporal (`wind`) dimension; Bureau of Meteorology (BoM) provides climate observations that are widely applied in agriculture and ecology study; air pollution data.

Our new data structure for spatio-temporal data

This paper describes the implementation of a new spatio-temporal data structure: `cubble`. `cubble` implements a relational data structure that uses two forms to manage the switch between spatial and temporal dimension. With this structure, users can manipulate the spatial or temporal dimension separately, while leaves the linking of two dimensions to `cubble`. The software is available from the Comprehensive R Archive Network (CRAN) at [CRAN link].

Section division

The rest of the paper will be divided as follows: [complete when the paper structure is more solid]

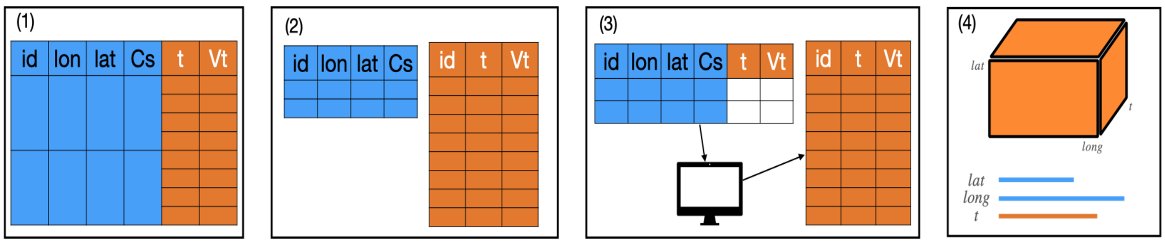


Figure 1: Illustration of incoming data formats for spatio-temporal data. (1) Data comes in as a single table; (2) Separate tables for spatial and temporal variables; (3) A single table with all the parameters used to query the database and a separate table for queried data; and (4) Cubical data in array or NetCDF format.

2. The cubble package

Spatio-temporal data usually come in various forms and Figure 1 shows four examples of this. No matter whichever form the data is in, there are always some common components shared by these data. A spatial identifier (`id` in the diagram) identifies each site. The temporal identifier (`t` in the diagram) [. . .]. Coordinates, comprising of latitude and longitude, are commonly used variables for point pattern data.These identifiers will be the building blocks for the data structure introduced below. For other variables, those invariant at each time stamp are spatial variables and those differ are temporal variables.

In a cubble, there are two forms: 1) nested form, for manipulating the spatial dimension, and 2) long form, for manipulating the temporal dimension. Figure 2 sketches the two forms along with the associated attributes. A variable identifies by the spatial identifies can come from manipulating spatial variables itself, or summary of temporal variables. The nested cubble is best suited to work with this type of operation, since it defines each spatial unit as a row. The spatial variables are directly displayed in columns. Temporal variables are nested in a column called `ts` and the underlying rowwise dataframe uses a `group` attributes to ensure each row is in its own group.

Temporal operations are suited to be performed in the long cubble as each row is defines as the combination of spatial and temporal identifier. This is also the structure that `tsibble` adopts. Temporal variables are directly displayed. To avoid repeating the same spatial at each temporal unit, all the spatial variables, along with the spatial identifier, are stored as a `spatial` attributes. This information is used when switching back to the nested form.

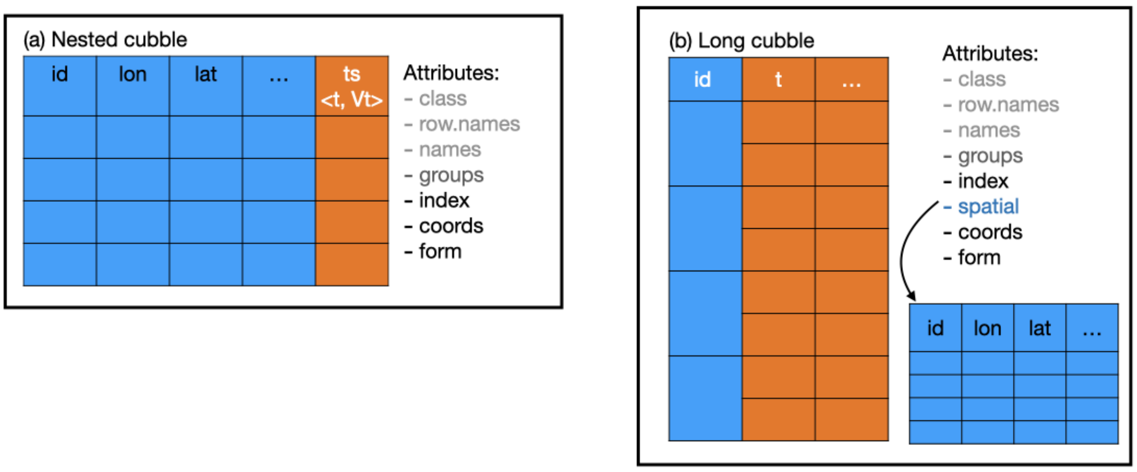


Figure 2: Illustration of nested and long cubble.

2.1. Create a cubble in the nested form

To use functionalities from cubble, data analysts first need to create a cubble. `as_cubble` create a `cubble` by supplying the three key components: `key` as the spatial identifier; `index` as the temporal identifier; and a vector of `coords` in the order of longitude first and then latitude. The use of `key` and `index` follows the naming convention in `tsibble`. The cubble created by default is in the nested form. Below is an example of creating a cubble:

```
R> (cubble_nested <- cubble::climate_flat %>%
+   as_cubble(key = id, index = date, coords = c("long", "lat")))

# cubble:   id [5]: nested form
# bbox:     [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
  id             lat  long  elev name            wmo_id ts
  <chr>        <dbl> <dbl> <dbl> <chr>            <dbl> <list>
1 ASN00009021 -31.9   116.   15.4 perth airport   94610 <tibble [366 x 4]>
2 ASN00010311 -31.9   117.  179   york            94623 <tibble [366 x 4]>
3 ASN00010614 -32.9   117.  338   narrogin        94627 <tibble [366 x 4]>
4 ASN00014015 -12.4   131.   30.4 darwin airport  94120 <tibble [366 x 4]>
5 ASN00015131 -17.6   134.  220   elliott         94236 <tibble [366 x 4]>
```

In the `cubble` header, you can read the name of the `key` variable, bbox, and also the name of variable nested in the `ts` column. In this example, the spatial identifier is `id` and the number in the bracket means there are 5 unique `id` in this dataset. The bbox in the second row gives the range of the coordinates. The temporal variables are all nested in the `ts` column, but it could be useful to know the name these variables. The third row in the cubble header shows these names and in this example this includes: precipitation, `prcp`, maximum temporature, `tmax`, and minimum temperature, `tmin`.

2.2. Stretch a nested cubble into the long form

From a created cubble in the nested form, analysts may want to directly manipulate the temporal variables. This would require switching to the long form using `stretch()`. The verb `stretch()` switch the cubble from the nested form into a long form. Under the hood, it first extracts the spatial variables into a separate tibble to store in the attribute `spatial` and then unnests the `ts` column to show the temporal content:

```
R> (cubble_long <- cubble_nested %>% stretch(ts))

# cubble:  date, id [5]: long form
# bbox:    [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# spatial: lat [dbl], long [dbl], elev [dbl], name [chr], wmo_id [dbl]
   id          date        prcp  tmax  tmin
   <chr>       <date>     <dbl> <dbl> <dbl>
 1 ASN00009021 2020-01-01     0  31.9  15.3
 2 ASN00009021 2020-01-02     0  24.9  16.4
 3 ASN00009021 2020-01-03     6  23.2  13
 4 ASN00009021 2020-01-04     0  28.4  12.4
 5 ASN00009021 2020-01-05     0  35.3  11.6
 6 ASN00009021 2020-01-06     0  34.8  13.1
 7 ASN00009021 2020-01-07     0  32.8  15.1
 8 ASN00009021 2020-01-08     0  30.4  17.4
 9 ASN00009021 2020-01-09     0  28.7  17.3
10 ASN00009021 2020-01-10     0  32.6  15.8
# ... with 1,820 more rows
```

Notice here that the third line in the header is changed to reflect the spatial variables stored. This is a format suitable for computing time-wise variables.

2.3. Tamp a long cubble back to the nested form

Manipulation on the spatial and temporal dimension can be an iterative process. Many times, we may decide to go back to the nested form after some temporal manipulation. The verb to switch a long cubble back to the nested form is `tamp()`:

```
R> (cubble_back <- cubble_long %>% tamp())

# cubble:   id [5]: nested form
# bbox:     [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
  id            lat  long  elev name           wmo_id ts
  <chr>       <dbl> <dbl> <dbl> <chr>           <dbl> <list>
1 ASN00009021 -31.9  116.  15.4 perth airport   94610 <tibble [366 x 4]>
2 ASN00010311 -31.9  117. 179   york            94623 <tibble [366 x 4]>
3 ASN00010614 -32.9  117. 338   narrogin        94627 <tibble [366 x 4]>
4 ASN00014015 -12.4  131.  30.4 darwin airport  94120 <tibble [366 x 4]>
5 ASN00015131 -17.6  134. 220   elliott         94236 <tibble [366 x 4]>
```

2.4. Migrate spatial variables to a long cubble

As an output to be supplied to further visualisation or modelling, analysts would usually like the spatial and temporal variables to be in the same table. `migrate()` moves the spatial variables in the attribute `spatial` into the long form cubble.

```
R> (cubble_long %>% migrate(long, lat))

# cubble:  date, id [5]: long form
# bbox:    [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# spatial: lat [dbl], long [dbl], elev [dbl], name [chr], wmo_id [dbl]
   id          date        prcp  tmax  tmin  long   lat
   <chr>       <date>     <dbl> <dbl> <dbl> <dbl> <dbl>
 1 ASN00009021 2020-01-01     0  31.9  15.3  116. -31.9
 2 ASN00009021 2020-01-02     0  24.9  16.4  116. -31.9
 3 ASN00009021 2020-01-03     6  23.2  13    116. -31.9
 4 ASN00009021 2020-01-04     0  28.4  12.4  116. -31.9
 5 ASN00009021 2020-01-05     0  35.3  11.6  116. -31.9
 6 ASN00009021 2020-01-06     0  34.8  13.1  116. -31.9
 7 ASN00009021 2020-01-07     0  32.8  15.1  116. -31.9
 8 ASN00009021 2020-01-08     0  30.4  17.4  116. -31.9
 9 ASN00009021 2020-01-09     0  28.7  17.3  116. -31.9
10 ASN00009021 2020-01-10     0  32.6  15.8  116. -31.9
# ... with 1,820 more rows
```

2.5. Support on hierarchical structure

`switch_key()`

2.6. Support on interactive graphics

2.7. Integrating into a tidy workflow

Building from an underlying `tbl_df` structure, it is natural to implement methods available in `dplyr` to `cubble`. Supported methods in the `cubble` with `dplyr` generics includes:

mutate	
filter	
summarise	
select	
arrange	
rename	
left_join	
group_by	
ungroup	
slice family	**slice_head**, **slice_tail**, **slice_sample**, **slice_min** and **slice_max**

`cubble` is also compatible with `tsibble` in the sense that the original list-column can be a `tbl_ts` object. Duplicates and gaps shoudl be first checked before structuring the data into a cubble. If the input data is a `tsibble` object, the long form cubble is also a `tsibble` where users can directly apply time series operations.

3. Examples

3.1. Australia precipitation pattern in 2020

Forming a cubble + basic tidyverse verbs - Vig 2 Aggregation - Vig 4

This vignette introduces how to perform spatial and temporal manipulate in a cubble with dplyr verbs. We will illustrate with `weatherdata::historical_tmax` data, which have the historical maximum temperature recorded for Australian stations with the earliest dating back to 1859.

Spatial manipulation

`historical_tmax` is already in a nested cubble format, which is suitable for station-wise manipulation. `rnoaa` construct the station id by prefix `ASN00` to the Bureau of Meteorology (BOM) station number. In BOM's numbering system, the 2nd and 3rd digit denotes the state a station is located in and this is equivalent to the 7th to 8th digit in our string. We can mutate/ filter station in a particular state based on this information and here we add a column called `state_id` and filter out the ones in New South Wales and Victoria (ranging from 46 to 90).

```
R> tmax_nested <- weatherdata::historical_tmax %>%
+   mutate(state_id = stringr::str_sub(id, 7, 8)) %>%
+   filter(between(state_id, 46, 90))
```

Temporal manipulation

There are some operations in the time dimension we would like to make:

- Extract observations in a particular period, say, those in 1971 to 1975 and 2016 to 2020. This can be used to compare the historical and recent climate.
- Summarise daily records into monthly to remove sparsity

These time dimension operations can be computed in the long form and `stretch()` converts a nested cubble to a long cubble. From a long cubble, we can write the exact same dplyr codes to complete the two tasks:

```
R> tmax_long <- tmax_nested %>%
+   stretch() %>%
+   filter(lubridate::year(date) %in% c(1971:1975, 2016:2020)) %>%
+   mutate(
+     month = lubridate::month(date),
+     group = as.factor(ifelse(
+       lubridate::year(date) > 2015,
+       "2016 ~ 2020",
+       "1971 ~ 1975"
+     ))
+   ) %>%
+   group_by(month, group) %>%
+   summarise(tmax = mean(tmax, na.rm = TRUE))
```

Back to spatial

A data quality issue with the `rnoaa` data is that while it records the first and last year recorded of each series, it doesn't report the period of missingness. For example, station `ASN00047048` starts it first record in 1957, pauses for a period from 1963 to 1990, and then resumes it recording till today. Out of the 77 stations in New South Wales and Victoria, 7 stations have this issue and we would like to remove those stations from the comparison.

Again, this is a station-wise operation and to convert back from the long cubble to a nested one, use `tamp()`. Here we keep the stations with 24 observations (12 months for both periods) after the monthly aggregation.

```
R> tmax_nested2 <- tmax_long %>%
+   tamp() %>%
+   filter(nrow(ts) == 24)
```

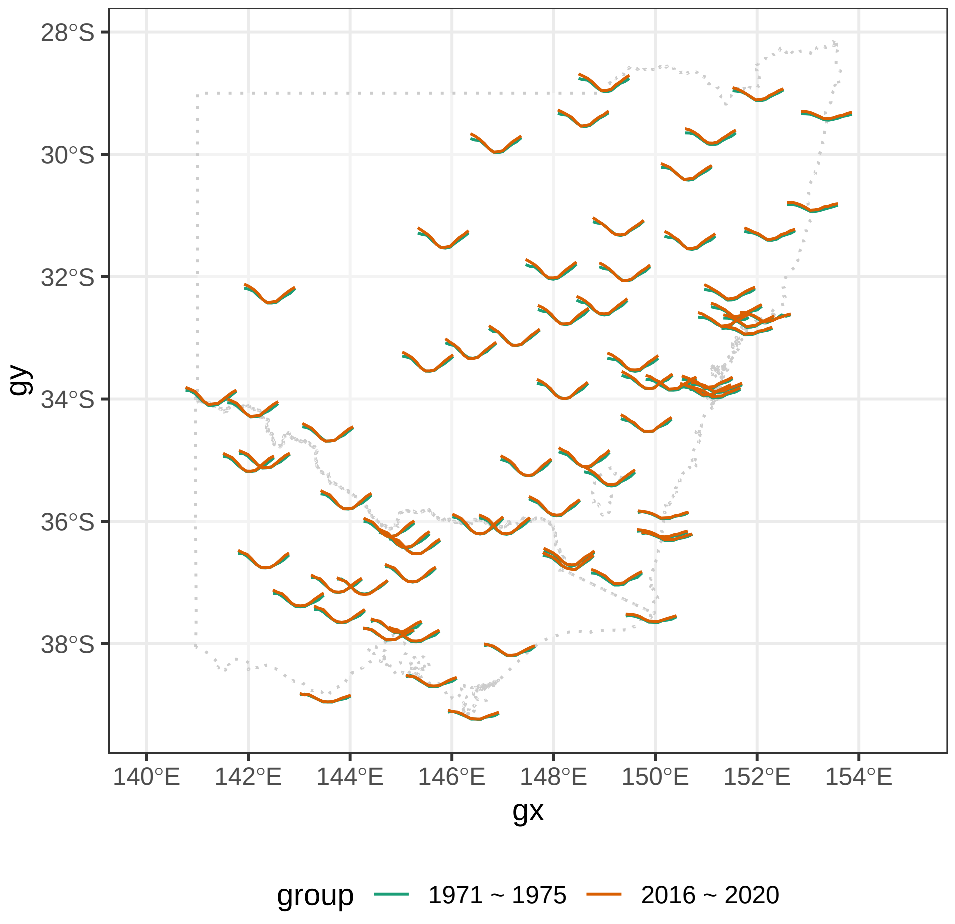
In some visualisation, we may need information from both spatial and temporal dimension. One example of this is a glyph map, where spatial variables, i.e. `longitude` and `latitude`, are used to construct the major axes and temporal variables, i.e. `month` and `tmax`, are used to construct the minor axes. This requires these variables to be in the same table, rather than in different forms. In cubble, you can append spatial variables that are invariant to the `key` withe `migrate()`

```
R> tmax_long2 <- tmax_nested2 %>%
+   stretch() %>%
+   migrate(latitude, longitude)
```

Glyph map

Now its time for the glyph map!

```
R> gly <- glyphs(
+   tmax_long2,
+   x_major = "longitude",
+   y_major = "latitude",
+   x_minor = "month",
+   y_minor = "tmax",
+   height = 0.6,
+   width = 1
+ )
R>
R> nsw_vic <- ozmaps::abs_ste %>%
+   filter(NAME %in% c("Victoria", "New South Wales"))
R>
R> plot_map(nsw_vic, fill = "transparent") +
+   geom_path(data = gly, aes(gx, gy,
+     group = interaction(id, group),
+     color = group
+   )) +
+   scale_color_brewer(palette = "Dark2") +
+   theme_bw() +
+   coord_sf(xlim = c(140, 155)) +
+   theme(legend.position = "bottom")
```



3.2. Spatial and temporal aggregation

On top of `aus_climate` data in this package, `climate_full` from package weatherdata has daily climate data of 639 Australia stations from 2016 to 2020. This is where these stations locate in an Australia map:

This is a lot of stations to look at for one climate variable and they can't all fit into a glyph map. What we can do is to group stations into clusters and look at the aggregated series in the glyph map. In this vignette, I will introduce how to perform spatial (and temporal) aggregation using hierarchical data structure in cubble.

First let's add a new column `ll` for calculating distance matrix in the next section and aggregate the daily precipitation into weekly measure. These steps should be familiar from the vignette *Data manipulation with cubble*.

```
R> station_nested <- weatherdata::climate_full %>%
+   mutate(ll = s2::s2_lnglat(long, lat)) %>%
+   stretch() %>%
+   mutate(wk = lubridate::week(date)) %>%
+   group_by(wk) %>%
+   summarise(prcp = sum(prcp, na.rm = TRUE)) %>%
+   tamp()
```

Hierarchical data structure

Imposing a clustering structure can be thought of as building a hierarchical structure where stations are nested within clusters. As an example to illustrate here, we use a kmean clustering algorithm based on the distance matrix and specify the number of centers to be 20. More complex algorithms can also be used for more complex problem, as long as a mapping from each station id to the cluster id can be constructed. We then join this data to our station data:

```
R> dist_raw <- scale(s2::s2_distance_matrix(station_nested$ll, station_nested$ll))
R>
R> cluster_res <- tibble(
+   id = station_nested$id,
+   cluster = kmeans(dist_raw, centers = 20, nstart = 50)$cluster
+ )
R>
R> station_nested <- station_nested %>%
+   left_join(cluster_res)
```

One thing we hope to do with the cluster is to find the coordinates of the centroid. These are variables variant to the station but invariant to the cluster and it would be nice to have a function that structure each cluster as a row. `switch_key()` is the function that does this: it lets you to specify a new key, say `cluster` and nests all spatial variables variant to `cluster` into a column. Temporal observations from different stations while within the same cluster are bound in the nested column `ts`.

```
R> cluster_nested <- station_nested %>%
+   switch_key(cluster)
```

This structure makes it easy to compute cluster level variable, although there are a few steps to calculate the centroid coordinates: we need to find the convex hull that wraps around the cluster, make it a polygon, find the centroid of the polygon and finally, extract the x and y coordinate of each centroid:

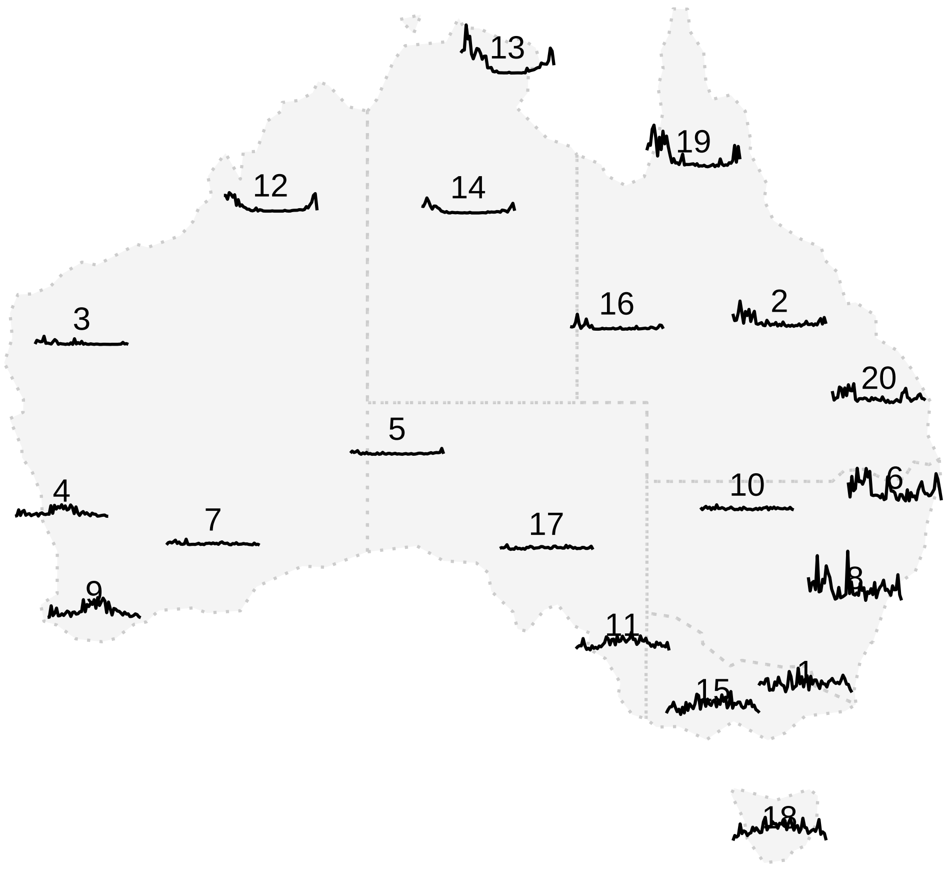
```
R> cluster_nested <- cluster_nested %>%
+   mutate(
+     chull = list(chull(.val$long, .val$lat)),
+     ll_cluster = sf::st_as_sfc(
+       s2::s2_make_polygon(c(.val$long[chull]),
+         c(.val$lat[chull]),
+         oriented = FALSE
+       )
+     ),
+     centroid = s2::s2_centroid(ll_cluster),
+     cent_long = s2::s2_x(centroid),
+     cent_lat = s2::s2_y(centroid)
+   )
```

After we have got `cluster_nested`, spatial and temporal data at both levels can be easily obtained. If we use `station` and `cluster` prefix to denote the two levels and `nested` and `long` for whether the data shows the spatial or temporal dimension, the relationship among the four datasets can be illustrated in the following workflow:

Start with the original `station_nested`, `stretch()` expands the `ts` column with each station (`id`) forming a group and attach variables invariant to `id` as an attribute. `switch_key()` changes the `key` from `id` to `cluster` and nests all the spatial variables that variant to `cluster`. `stretch()` `cluster_nested` will store variables that are invariant to `cluster` as a tibble in the attribute.

Now we can obtain the aggregated series as described in the workflow diagram above and construct the glyph map with `GGally::glyphs()`:

```
R> cluster_long <- cluster_nested %>%
+   stretch(ts) %>%
+   group_by(wk) %>%
+   summarise(prcp = sum(prcp, na.rm = TRUE)) %>%
+   migrate(cent_long, cent_lat)
R>
R> gly <- GGally::glyphs(cluster_long,
+   x_major = "cent_long", x_minor = "wk",
+   y_major = "cent_lat", y_minor = "prcp",
+   height = 2, width = 4
+ )
R>
R> state_map <- rmapshaper::ms_simplify(ozmaps::abs_ste, keep = 2e-3)
R> plot_map(state_map) +
+   geom_text(
+     data = cluster_nested,
+     aes(x = cent_long, y = cent_lat, label = cluster)
+   ) +
+   geom_path(
+     data = gly,
+     aes(x = gx, y = gy, group = gid)
+   )
```



We can also look at the precipitation of each individual station within the same cluster:

```
R> station_long <- station_nested %>%
+   stretch(ts) %>%
+   migrate(cluster)
R> station_long %>%
+   ggplot(aes(x = wk, y = prcp, group = id)) +
+   geom_line(alpha = .3) +
+   facet_wrap(vars(cluster), scales = "free_y", ncol = 4) +
+   theme_bw()
```

Lastly, there is one series on Tasmania island standing out from others, lets look at where it is:

```
R> # this part still needs some fixing
R> tas_latlong <- station_nested %>%
+   filter(lat < -40) %>%
+   mutate(p = max(ts$prcp)) %>%
+   strip_rowwise() %>%
```

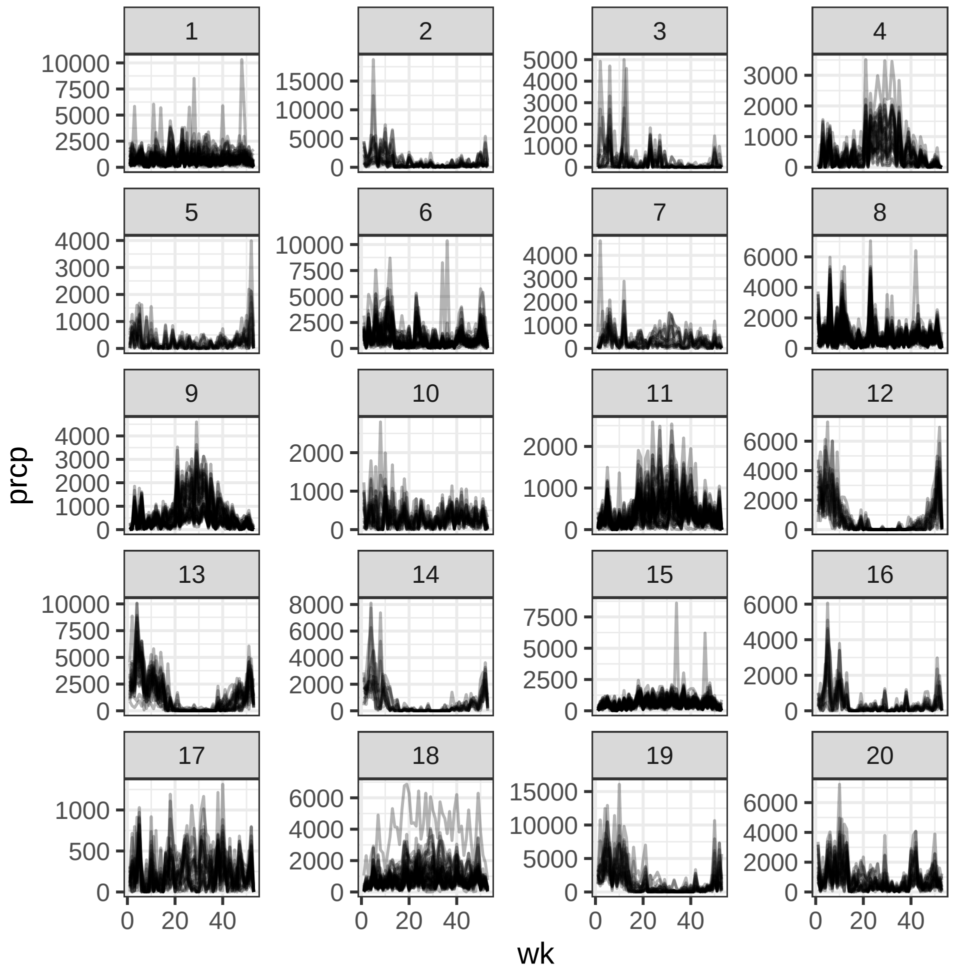
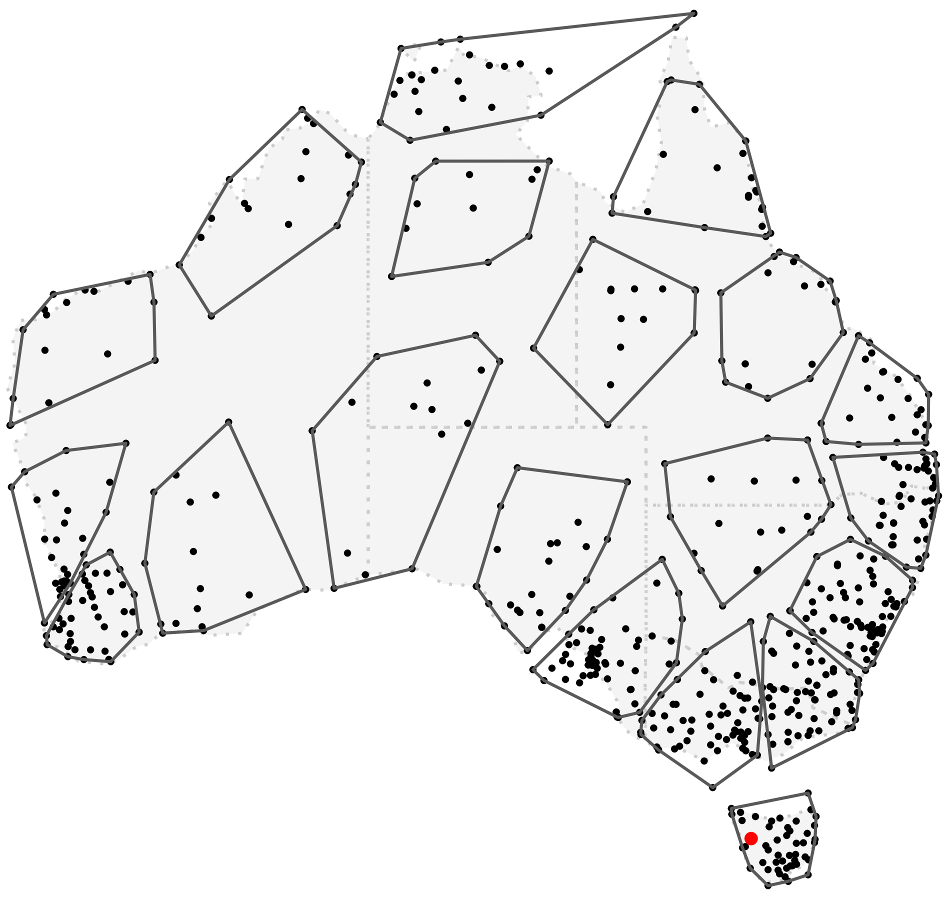


Figure 3: shtishisa

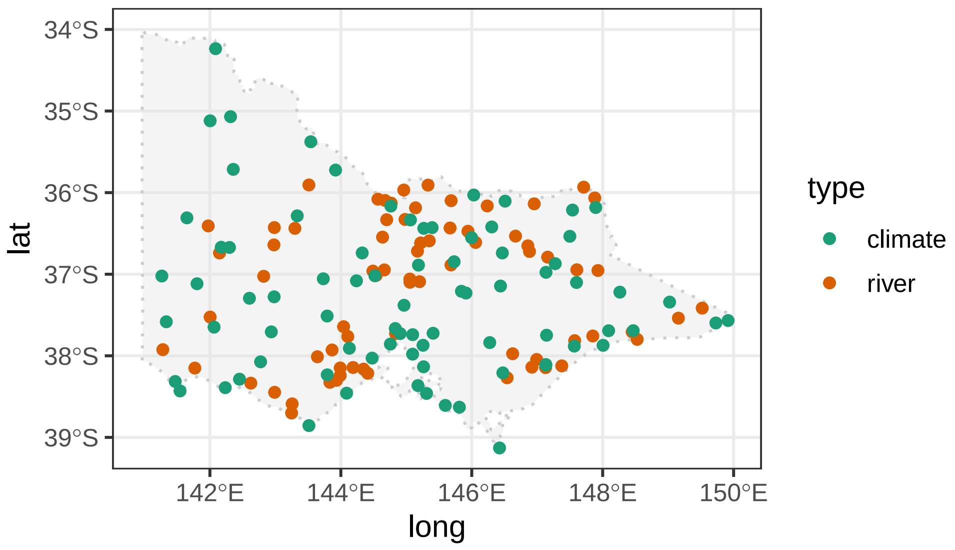
```
+   filter(p == max(p))
R>
R> state_map <- rmapshaper::ms_simplify(ozmaps::abs_ste, keep = 2e-3)
R> plot_map(state_map) +
+   geom_point(data = station_nested, aes(x = long, y = lat), size = 0.5) +
+   geom_sf(data = cluster_nested, aes(geometry = ll_cluster), fill = "transparent") +
+   geom_point(data = tas_latlong, aes(x = long, y = lat), col = "red")
```



3.3. Matching precipitation and river level in Victria water gauges

The water level data comes from Bureau of Meteorology and has a copy in `weatherdata`. Here we extract the water course level and add a column annotate this data of type `river`. For the rainfall data, we will still use the `weatherdata::climate_full`, filtering for Victorian stations in 2020 should be pretty familiar by now. Again, we first look at where these stations are on the map first:

```
R> river <- weatherdata::water %>%
+   stretch() %>%
+   select(date, Water_course_level) %>%
+   tamp() %>%
+   mutate(type = "river")
R>
R> climate <- weatherdata::climate_full %>%
+   filter(between(stringr::str_sub(id, 7, 8), 76, 90)) %>%
+   stretch() %>%
+   filter(lubridate::year(date) == 2020) %>%
+   tamp() %>%
+   mutate(type = "climate")
R>
R> vic_map <- rmapshaper::ms_simplify(ozmaps::abs_ste %>% filter(NAME == "Victoria"))
R> plot_map(vic_map) +
+   geom_point(
+     data = dplyr::bind_rows(river, climate),
+     aes(x = long, y = lat, color = type)
+   ) +
+   scale_color_brewer(palette = "Dark2") +
+   theme_bw()
```



Theory

Temporal matching checks how spatially matched pairs align temporally. We use the following chart to illustrate how the temporal matching works:

For each spatially matched pair, say `A` and `a`, we first find the largest `n` points in each series, colored in brown points here. Here we use the largest three but you can tune this number by `temporal_n_highest`. Then we construct the interval of the largest points from one series and see how many points, from the other series, fall into the intervals. The series used to construct the interval is controlled by `temporal_independent` and the window size by `temporal_window` with a default of 5.

In this illustration, we construct the interval based on series `A` and two of the three peaks from `a` falls into this interval at Time 7 and 27.

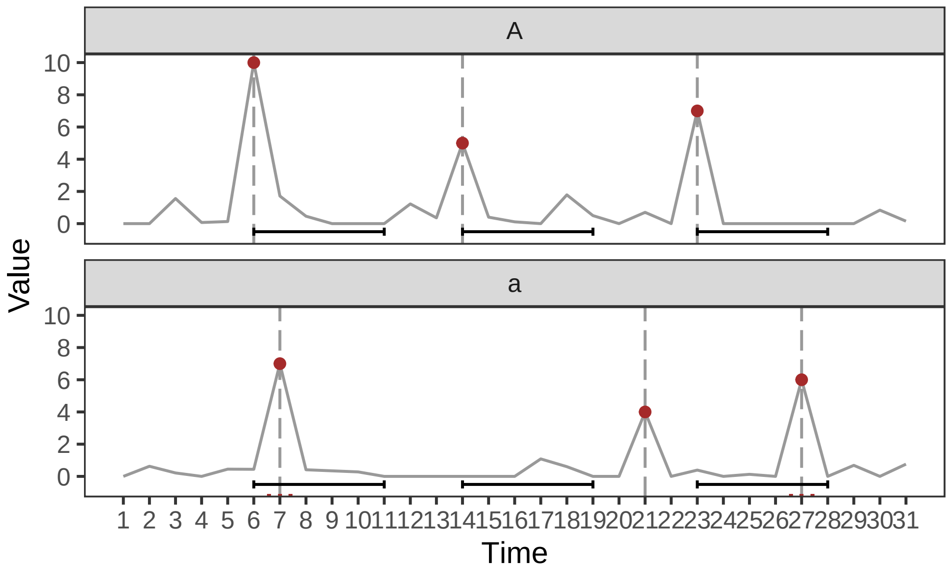


Figure 4: shtishisaasdf

Rainfall translates into river level

There's another mandatory argument that hasn't been introduced above: `temporal_var_to_match`. This argument controls the variable to match and it needs to appear in both the `major` and `minor` set. In the water level matching example, we match the variable `Water_course_level` from `river` to `prcp` from `climate`, hence need to manually rename one of them to match the other, here we rename `Water_course_level` to `prcp` in `river`:

```
R> river <- river %>%
+   stretch() %>%
+   rename(prcp = Water_course_level) %>%
+   tamp()
```

Now we use `match_sites()` to first pair the weather stations with the river gauges spatially and then apply the temporal matching on `prcp`. We will construct the interval based on peaks in `climate` since we would expect a lag effect for precipitation to flow into the river and cause a raise in river level, hence `temporal_independent = climate`. We select the 30 highest peak from the series to construct the match by setting `temporal_n_highest = 30`. This is a tuning parameter and you can start with 10% of the points of one series (here we have daily data for a year, 10% is roughly 30 points). `temporal_min_match` filters out pairs don't have enough match and to return all the pairs, set `temporal_min_match` to `0`.

```
R> res <- match_sites(river, climate,
+   temporal_var_to_match = prcp,
+   temporal_independent = climate,
+   temporal_n_highest = 30,
+   temporal_min_match = 15
+ )
```

```
R>
R> res

# cubble:   id [8]: nested form
# bbox:     [144.52, -37.73, 146.06, -36.55]
# temporal: date [date], prcp [dbl]
  id          name                 lat  long type  ts         .dist .group n_match
  <chr>       <chr>              <dbl> <dbl> <chr> <list>     <dbl>  <int>   <int>
1 405234      SEVEN CREEKS @ D~ -36.9  146. river <tibble ~  6.15      5      21
2 ASN00082042 strathbogie       -36.8  146. clim~ <tibble ~  6.15      5      21
3 404207      HOLLAND CREEK @ ~ -36.6  146. river <tibble ~  8.54     10      21
4 ASN00082170 benalla airport   -36.6  146. clim~ <tibble ~  8.54     10      21
5 230200      MARIBYRNONG RIVE~ -37.7  145. river <tibble ~  6.17      6      19
6 ASN00086038 essendon airport  -37.7  145. clim~ <tibble ~  6.17      6      19
7 406213      CAMPASPE RIVER @~ -37.0  145. river <tibble ~  1.84      1      18
8 ASN00088051 redesdale         -37.0  145. clim~ <tibble ~  1.84      1      18
```

The output from temporal matching is also a cubble, with additional column `.dist` and `.group` inherent from spatial matching and `n_match` for the number of matched temporal peaks. Then you can use this output to plot the location of match or to look at the series:

```
R> p1 <- plot_map(vic_map) +
+   geom_point(
+     data = res,
+     aes(x = long, y = lat, color = type)
+   ) +
+   ggrepel::geom_label_repel(
+     data = res %>% filter(type == "river"),
+     aes(x = long, y = lat, label = .group)
+   ) +
+   scale_color_brewer(palette = "Dark2") +
+   ggtitle("Victoria") +
+   theme_minimal() +
+   theme(
+     panel.grid.major = element_blank(),
+     panel.grid.minor = element_blank(),
+     legend.position = "bottom"
+   )
R>
R> res_long <- res %>%
+   stretch(ts) %>%
+   migrate(.group, type) %>%
+   mutate(prcp = scale(prcp)[, 1]) %>%
+   switch_key(.group) %>%
+   migrate(type)
R>
R> p2 <- res_long %>%
```

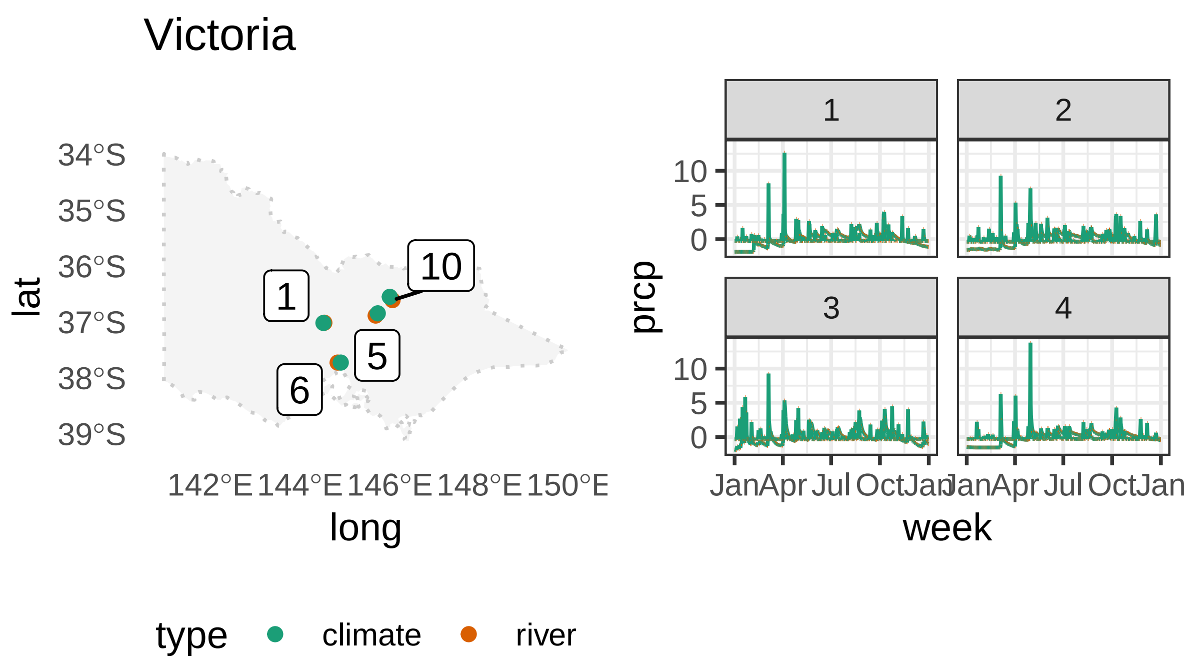


Figure 5: ...

```
+   ggplot(aes(
+     x = date, y = prcp,
+     group = id, color = type
+   )) +
+   geom_line() +
+   facet_wrap(vars(.group)) +
+   scale_color_brewer(palette = "Dark2", guide = "none") +
+   theme_bw() +
+   labs(x = "week") +
+   scale_x_date(date_labels = "%b")
R>
R> p1 | p2
```

3.4. Interative graphic with cubble

```
R> set.seed(123)
R> tmax_data <- weatherdata::climate_full %>%
+   slice_sample(n = 50) %>%
+   stretch() %>%
+   mutate(ym = tsibble::yearmonth(date)) %>%
+   group_by(ym) %>%
+   summarise(
+     tmax = mean(tmax, na.rm = TRUE),
+     tmin = mean(tmin, na.rm = TRUE),
+     prcp = sum(prcp, na.rm = TRUE)
+   ) %>%
+   ungroup(ym) %>%
+   mutate(
+     year = as.factor(year(ym)),
+     month = as.factor(month(ym)),
+     prcp = sqrt(prcp + 0.001)
+   ) %>%
+   tamp() %>%
+   mutate(median_tmax = median(ts$tmax, na.rm = TRUE)) %>%
+   stretch() %>%
+   migrate(median_tmax, long, lat, name) %>%
+   highlight_key(~id)
R>
R> p1 <- tmax_data %>%
+   ggplot(aes(
+     x = month, y = tmax, group = interaction(year, id),
+     color = median_tmax, label = name
+   )) +
+   geom_line() +
+   facet_wrap(vars(fct_reorder(name, -median_tmax))) +
+   theme(legend.position = "none")
R>
R> state_map <- rmapshaper::ms_simplify(ozmaps::abs_ste, keep = 2e-3)
R> p2 <- tmax_data %>%
+   ggplot() +
+   geom_sf(data = state_map, aes(geometry = geometry)) +
+   geom_point(aes(x = long, y = lat, color = median_tmax, label = name)) +
+   theme_void()
R>
R> bscols(
+   ggplotly(p1, width = 800, height = 800, tooltip = "label") %>%
+     highlight(on = "plotly_selected", off = "plotly_deselect", color = "red"),
+   ggplotly(p2, width = 800, height = 800, tooltip = "label") %>%
+     highlight(on = "plotly_selected", off = "plotly_deselect", color = "red")
+ )
```

4. Conclusion

5. Old stuff

Many spatial and spatio-temporal data structures have been developed by the R-spatial team for both raster and vector spatial data. For vector spatial data, which is the focus of this paper, `sf` (?) represents spatial vector information with simple features: points, lines, polygons and their multiples. Various `st_` function are designed to manipulate these features based on their geometric relationships. For spatio-temporal data, `stars` (Pebesma 2021) can represent both raster and vector data using multi-dimensional array. However, the underlying array structure can be difficult to operate for data analysts who are more familiar with a flat 2D data frame structure used by the tidyverse ecosystem.

In the temporal aspect, the `tsibble` (?) structure and its tidyverts ecosystem have provided a [. . .] workflow to work with temporal data. In a tsibble structure, temporal data is characterised by `index` and `key` where `index` is the temporal identifier and `key` is the identifier for multiple series, which could be used as a spatio identifier. However, a tsibble object, by construction, always requires the `index` in its structure. This makes it less appealing for spatio-temporal data since the output of calculated spatio-specific variables (i.e. features of each series) don't have the time dimension. Analysts will either need to have an additional step to join this output to the original tsibble or operate with variables stored in two separate objects. In addition, the long form structure of a tsibble object means spatio variables (i.e. longitude, latitude, and features of each series if joined back to the tsibble) of each spatio identifier will be repetitively recorded at each timestamp. This repetition is unnecessary and would inflate the object size for long series.

6. A new data structure for spatio-temporal data

The main difficulty and challenge
The main difficulty in visualising this type of data is to show information in both space and time dimension with the proper level of details without information overflow. This would sometimes require aggregating the time dimension into the proper level or slicing the data into a reasonable number of subset for display. In this sense, a data structure that regulates the manipulation spatio-temporal data will benefit the analysis workflow. While many implementations focus on manipulating and visualising pure spatial or temporal data, there are not sufficient tools to deal with spatio-temporal data. The purpose of this paper is to introduce a spatio-temporal vector data structure for data analysis in R.

To work with spatio-temporal data, analysts can choose to either work separately on each dimension or join the two sets together, however, each approach has its own problem: While is is natural to work separately on each sheet (since spatial and temporal operations usually don't overlap), analysts will need to manually keep the other data frame up to date. For example, the following pseudo code illustrates the scenario where once the spatial dataset is filtered for those within Victoria, the temporal dataset needs to be manually updated to reflect this spatial filter.

```
R> spatial_new <- spatial %>% filter(SITES_IN_VICTORIA)
R> temporal_new <- temporal %>% filter(id %in% spatial_new$id)
```

If analysts choose to join the spatial and temporal data together, the joined dataset could be too large since each spatial variable will be repeated at each time stamp for each site. Also, recordings of the site ID from different data sources can be slightly different from each other, causing a painful checking and cleaning of site IDs before the join.

7. Create a cubble

The creation of a cubble requires the site identifier (`key`), as well as the spatial (`coords`) and temporal (`index`) identifier. `climate_flat` is already a tibble and it uses `id` to identify each station, `date` as the time identifier, and `c(long, lat)` as the spatial identifier. To create a cubble for this data, use:

```
R> climate_flat %>% as_cubble(key = id, index = date, coords = c(long, lat))

# cubble:   id [5]: nested form
# bbox:     [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
  id            lat  long  elev name            wmo_id ts
  <chr>       <dbl> <dbl> <dbl> <chr>            <dbl> <list>
1 ASN00009021 -31.9  116.  15.4 perth airport    94610 <tibble [366 x 4]>
2 ASN00010311 -31.9  117. 179   york             94623 <tibble [366 x 4]>
3 ASN00010614 -32.9  117. 338   narrogin         94627 <tibble [366 x 4]>
4 ASN00014015 -12.4  131.  30.4 darwin airport   94120 <tibble [366 x 4]>
5 ASN00015131 -17.6  134. 220   elliott          94236 <tibble [366 x 4]>
```

Most of the time, spatio-temporal data doesn't come into this form and analysts need to query the climate variables based on station metadata. This is also a problem illustrated in Section 3.5 in @tidydata. Here we provide a structured way to query this data based on the row-wise operator and nested list. For this type of task, one can structure a metadata into a tibble and use row-wise operator to query the climate variables into a nested list. As an example here we demonstrate the workflow to find the 5 closest stations to Melbourne. We first create a station data frame with the 5 target stations.

```
# A tibble: 5 x 8
  id            lat  long  elev name                  wmo_id  dist city
  <chr>       <dbl> <dbl> <dbl> <chr>                  <dbl> <dbl> <chr>
1 ASN00086038 -37.7  145.  78.4 essendon airport       95866  10.8 melbourne
2 ASN00086282 -37.7  145. 113.  melbourne airport      94866  20.1 melbourne
3 ASN00086077 -38.0  145.  12.1 moorabbin airport      94870  21.9 melbourne
4 ASN00088162 -37.4  145. 528.  wallan (kilmore gap)   94860  48.1 melbourne
5 ASN00087113 -38.0  144.  10.6 avalon airport         94854  48.8 melbourne
```

We can query the climate information into a nested list named `ts` for each station with the `rowwise()` operator. To create a cubble, supply the same identifiers as with the first example.

```
R> sydmel_climate <- stations %>%
+   rowwise() %>%
+   mutate(ts = list(meteo_pull_monitors(id,
+     date_min = "2020-01-01",
+     date_max = "2020-12-31",
+     var = c("PRCP", "TMAX", "TMIN")
+   ) %>%
+     select(-id))) %>%
+   as_cubble(key = id, index = date, coords = c(long, lat))

# cubble:   id [5]: nested form
# bbox:     [144.47, -38.03, 145.1, -37.38]
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
  id            lat  long  elev name                  wmo_id  dist city   ts
  <chr>       <dbl> <dbl> <dbl> <chr>                  <dbl> <dbl> <chr>  <list>
1 ASN00086038 -37.7  145.  78.4 essendon airport       95866  10.8 melbo~ <tibbl~
2 ASN00086282 -37.7  145. 113.  melbourne airport      94866  20.1 melbo~ <tibbl~
3 ASN00086077 -38.0  145.  12.1 moorabbin airport      94870  21.9 melbo~ <tibbl~
4 ASN00088162 -37.4  145. 528.  wallan (kilmore gap)  94860  48.1 melbo~ <tibbl~
5 ASN00087113 -38.0  144.  10.6 avalon airport         94854  48.8 melbo~ <tibbl~
```

Below are the how the nested and long form look like for Australia climate data, which records daily precipitation, maximum and minimum temperature for 55 stations across Australia from 2015- 2020. Notice that each station forms a group in both forms and specifically, the nested and long form have a underlying `rowwise_df` and `grouped_df` respectively.

With a cubic framework on mind, different types of manipulation with cubble can be thought of as slicing the cube in various way. The table below shows how some `dplyr` verbs are mapped into the operation in a cubble. With the existing grouping on the station, additional groupping can be added with `group_by` and removed with `ungrouped`. [talk about why it is useful]

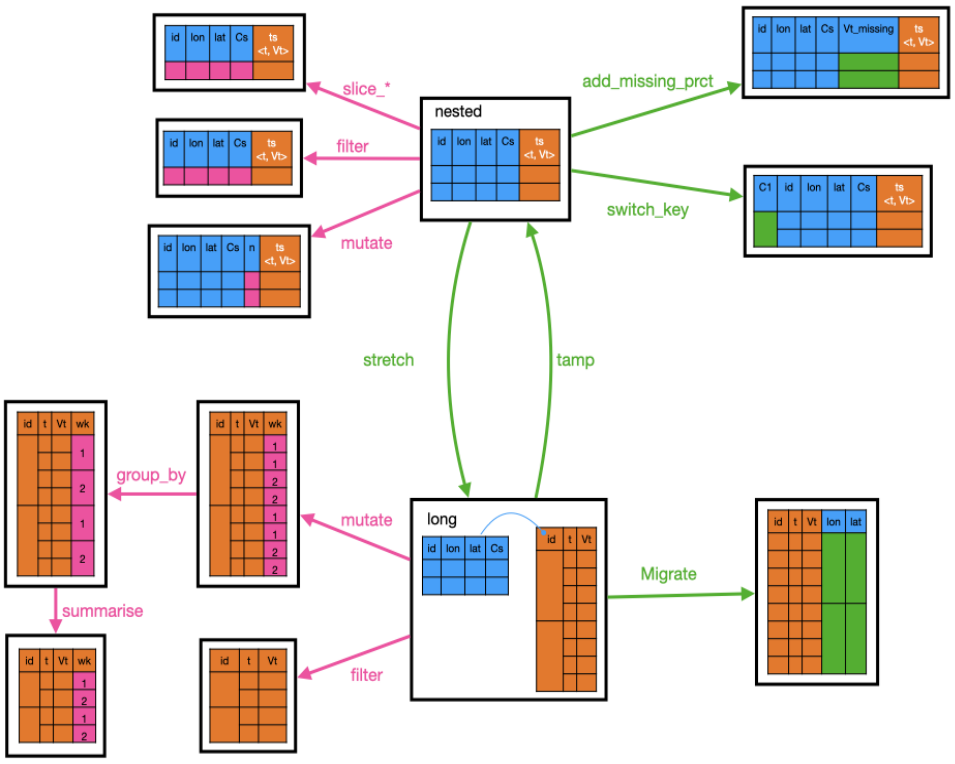


Figure 6: Cubble operations

7.1. Cubble operations

Basics

- `stretch`: nest to long form
- `tamp`: long to nest form
- `migrate`: move selected spatial variables to the long form.
- `add_dscrb_prct`: summary stats for missingness

dplyr compatibility:

- mutate, filter, summarise, select, arrange
- group and ungroup: group_by, ungroup
- slice family

Combine two cubbles

- match river and weather gauges data

- involve combining two cubbles
- join operations combine the two together by appending more rows but what we really want is to bind rows.
- bind rows also doesn't work since we want to bind only when there' s a matching????
- introduce bind_join

Hierarchical structure in cubble

- hierarchical is common.
- Given examples.
- Essence: switch between different levels
- introduce `switch_key`

8. Examples

Daily climate data (prcp, tmax, and tmin) from RNOAA - lots of stations across Australia

An exploratory data analysis questions: What's the climate profile look like in Australia

- General features: Any general trend/ fluctuation in prcp, tmax, and tmin?
- Local features: Any station stands out from the crowd?

References

Baddeley A, Turner R (2005). "Spatstat: An R Package for Analyzing Spatial Point Patterns." *Journal of Statistical Software*, **12**(6), 1–42. URL `https://doi.org/10.18637/jss.v012.i06`.

Pebesma E (2012). "spacetime: Spatio-Temporal Data in R." *Journal of Statistical Software*, **51**(7), 1–30. URL `https://doi.org/10.18637/jss.v051.i07`.

Pebesma E (2021). *stars: Spatiotemporal Arrays, Raster and Vector Data Cubes*. R package version 0.5-2, URL `https://CRAN.R-project.org/package=stars`.

Pebesma E, Bivand RS (2005). "S classes and methods for spatial data: the sp package." *R news*, **5**(2), 9–13.

Pebesma EJ (2018). "Simple features for R: standardized support for spatial vector data." *R Journal*, **10**(1), 439.

Ryan JA, Ulrich JM (2020). *xts: eXtensible Time Series*. R package version 0.12.1, URL `https://CRAN.R-project.org/package=xts`.

Wang E, Cook D, Hyndman RJ (2020). "A new tidy data structure to support exploration and modeling of temporal data." *Journal of Computational and Graphical Statistics*, **29**(3), 466–478. doi:10.1080/10618600.2019.1695624. URL https://doi.org/10.1080/10618600.2019.1695624.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. URL https://doi.org/10.18637/jss.v059.i10.

Wickham H (2020). *cubelyr: A Data Cube 'dplyr' Backend*. R package version 1.0.1, URL https://CRAN.R-project.org/package=cubelyr.

Affiliation:

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd