The GRASS GIS Temporal Framework: Object oriented code design with examples

Sören Gebbert
Thünen Institute of Climate-Smart Agriculture, Germany

Edzer Pebesma
Institute for Geoinformatics, University of Münster, Germany

This document is part of the supplementary material of the publication

The GRASS GIS Temporal Framework
to be published in
International Journal of Geographical Information Science in 2017

January 4, 2017

Contents

1 Introduction

This document describes the object oriented design of the GRASS GIS Temporal Framework that underlies the temporally enabled GRASS GIS, called TGRASS. TGRASS is a full featured field-based temporal GIS, see [Gebbert and Pebesma(2014)]. We provide several code examples to show the capabilities of the GRASS GIS Temporal Framework, ranging from spatio-temporal analysis of space-time dataset relations and their associated time stamped map layers, over topological relations between map layers, single pixel and feature access to the spatio-temporal intersection between trajectory data.

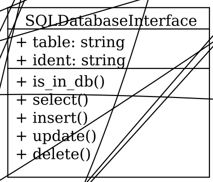
2 Object oriented code design

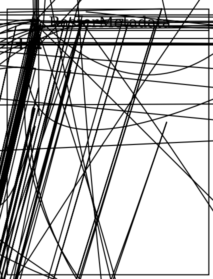
Objects that represent GRASS map layers and space-time datasets in the temporal framework are stored in the temporal database. The content of these objects can be serialised, since only serialisable Python types are used to represent the map layer and STDS metadata. The Unified Modelling Language (UML) diagram in figure 1 shows the simplified inheritance structure of the spatio-temporal extent and basic metadata classes of map layers and STDS. The metadata content is stored in an object specific Python dictionary. The class *SQLSerializer* takes care of serialising the dictionary content and creates select, insert, update and delete SQL statements that are used by its subclass *SQLDatabaseInterface* to manage metadata in the temporal database. This class also specifies the unique identifier for map layers and space-time datasets. It is a combination of the map layer name or space-time datasets name and the associated mapset. The identifiers for vector layer may contain an additional reference to an attribute table, since a vector layer may have several different attribute tables. The temporal framework allows time stamps for attribute tables. Hence, a single vector layer with multiple attribute tables may have several time stamps and therefore several entries in the temporal database. The class *DatasetBase* inherits from *SQLDatabaseInterface* and specifies basic information about map layer and space-time datasets. Its subclass *STDSBase* specifies in addition the modification time for space-time datasets. The class *TemporalExtent* inherits from *SQLDatabaseInterface* and specifies the temporal extent for map layers and space-time datasets. It provides methods to compute temporal relationships, temporal intersections, unions and disjoint unions between itself and a second object of type *TemporalExtent*. Absolute and relative time temporal extents are specified in subclasses from which space-time dataset specific classes derive that specify the temporal granularity and the temporal type of the registered maps. The spatial extent of map layers and space-time datasets is represented by the class *SpatialExtent*. It inherits from *SQLDatabaseInterface* and supports two-dimensional[1] and three-dimensional spatial extents[2]. Similar to the temporal extent, the class *SpatialExtent* provides methods for the computation of spatial relationships, spatial intersections, unions and disjoint unions between itself and a second object of type *SpatialExtent* in two and three dimensions.

[1]north, south, east, west

[2]north, south, east, west, top, bottom

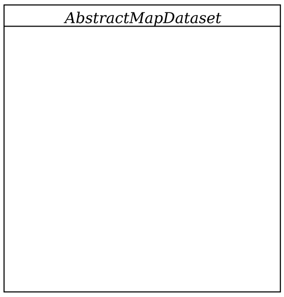
aggregate this data into minimum and maximum values of all registered map layer minimums and minimum and maximum values of all registered map layer maximums.





for map layers and space-time datasets. It provides methods for spatial and temporal operations, whereby the spatial methods are specified as abstract methods that must be implemented in subclasses that define a two- or three-dimensional spatial extent. The *AbstractDataset* class is derived from two parent classes, *SpatialTopologyDatasetConnector* and *TemporalTopologyDatasetConnector*. These classes provide the functionality to connect *AbstractDataset* objects to build a spatio-temporal topology structure. To actually build the spatio-temporal topology a dedicated class, *SpatioTemporalTopologyBuilder*, was implemented. This class builds the spatio-temporal topology of an unordered list of *AbstractDataset* objects, or between two unordered lists of *AbstractDataset* objects using their connector functionality. The *SpatioTemporalTopologyBuilder* requires the GRASS GIS vector library R*-Tree that is used for topological vector operations. This library is implemented in C and was improved to allow the fast and memory efficient creation of spatio-temporal topology structures.

Raster, 3D raster and vector layer are represented in the temporal framework by the *AbstractMapDataset* class, a subclass of *AbstractDataset*. It implements methods to set the temporal extent and to buffer the temporal and spatial extents. Map layers can be registered in several different space-time datasets at the same time, hence they must keep track in which space-time datasets they are registered to perform coherent deletion and unregistration operations. This is assured by the methods *get_*


AbstractMapDataset

The last example in listing 4 shows how to use the GRASS GIS Temporal Framework to perform spatio-temporal intersections between trajectory data. The input data of time stamped GPS coordinates from two Zebras in northern Botswana was provided by [Bartlam-Brooks and Harris(2014)]. About 13000 single GPS positions are available in two comma separated value (csv) files, one for each zebra. The spatio-temporal coordinates from the csv files are used directly without the creation of intermediate vector layers. The spatio-temporal locations of each Zebra is buffered in space and snapped in time to allow spatio-temporal intersection computation. The result is filtered to detect only meaningful events in which the two animals may have met.

Listing 5 shows the Python code that was used to perform benchmark runs for typical use cases of the temporal framework.

3.1 Registration of existing raster layers

Listing 1: **Import and registration of European daily mean temperature raster layers from 1950 - 1964** from the gridded (EOB-S) European Climate and Assessment Dataset (ECAD) by [Haylock et al.(2008)Haylock, Hofstra, Tank, Klok, Jones, and New]. We use a dedicated temporal database connection interface to speedup the database access. The GRASS module *r.in.gdal* is used to import the netCDF file with temperature time series raster data. This module will create 5479 new raster layer in the current mapset. The numbering scheme is *temperature_daily.[1-5479]*. It is used to create *RasterDataset* objects that are inserted into the temporal database and registered in a new created STRDS. Finally the metadata of the new STRDS is updated and printed to stdout.

```python
#!/usr/bin/env python
from datetime import datetime
from datetime import timedelta
from grass.pygrass import modules
import grass.temporal as tgis

tgis.init()

dbif = tgis.SQLDatabaseInterfaceConnection()
dbif.connect()

modules.Module("r.in.gdal", input="tg_0.25deg_reg_1950-1964_v10.0.nc",
               output="temperature_daily", flags="o", overwrite=True)


temperature = tgis.open_new_stds(name="temperature_daily",
                                 type="strds",
                                 temporaltype="absolute",
                                 title="Daily mean temperature Europe",
                                 descr="Daily mean temperature of Europe 1950 - 1964",
                                 semantic="mean",
                                 overwrite=True, dbif=dbif)

for day in xrange(5479):
    temperature_id = "temperature_daily.%i@%s"%(day + 1, tgis.get_current_mapset())

    temperature_layer = tgis.RasterDataset(temperature_id)
    temperature_layer.load()

```

```
    start = datetime(1950, 1, 1) + timedelta(day)
    end = datetime(1950, 1, 1) + timedelta(day + 1)

    temperature_layer.set_absolute_time(start, end)

    if temperature_layer.is_in_db(dbif=dbif) is False:
        temperature_layer.insert(dbif=dbif)
    else:
        temperature_layer.update(dbif=dbif)

    temperature.register_map(map=temperature_layer, dbif=dbif)

temperature.update_from_registered_maps(dbif=dbif)
temperature.print_info()

dbif.close()
```

Running the script will result in the following output:

```
"""
 +-------------------- Space-Time Raster Dataset -------------------------+
 |                                                                        |
 +-------------------- Basic information ---------------------------------+
 | Id: ........................ temperature_daily@soeren
 | Name: ...................... temperature_daily
 | Mapset: .................... soeren
 | Creator: ................... soeren
 | Temporal type: ............. absolute
 | Creation time: ............. 2014-06-11 18:55:21.727892
 | Modification time:.......... 2014-06-11 18:56:41.223508
 | Semantic type:.............. mean
 +-------------------- Absolute time -------------------------------------+
 | Start time:................. 1950-01-01 00:00:00
 | End time:................... 1965-01-01 00:00:00
 | Granularity:................ 1 day
 | Temporal type of maps:...... interval
 +-------------------- Spatial extent ------------------------------------+
 | North:...................... 75.5
 | South:...................... 25.25
 | East:.. .................... 75.5
 | West:....................... -40.5
 | Top:........................ 0.0
 | Bottom:..................... 0.0
 +-------------------- Metadata information ------------------------------+
 | Raster register table:......
     raster_map_register_5c8a728f003a4bf7b676c1b26bb91e6b
 | North-South resolution min:. 0.25
 | North-South resolution max:. 0.25
 | East-west resolution min:... 0.25
 | East-west resolution max:... 0.25
 | Minimum value min:.......... -4961.0
 | Minimum value max:.......... 517.0
 | Maximum value min:.......... 969.0
 | Maximum value max:.......... 3704.0
 | Aggregation type:........... None
 | Number of registered maps:.. 5479
 |
 | Title:
 | Daily mean temperature Europe
 | Description:
 | Daily mean temperature of Europe 1950 - 1964
 | Command history:
 | # 2014-06-11 18:55:21
 | import_register.py
 |
 +------------------------------------------------------------------------+
"""
```

3.2 Temporal aggregation of a raster time series

Listing 2: **Aggregation of daily mean temperature to monthly mean temperature.** To guarantee correct aggregation the region of the current mapset is set to the daily mean temperature map layer extent and resolution using the GRASS module *g.region*. The daily STRDS is opened and a new STRDS that will manage the monthly aggregated data is created. We use a dedicated temporal database connection interface to speed-up the database access. A list of temporary *RasterDataset* objects is created that have monthly time stamps. This list is used together with the daily map layer object list to compute the temporal topology between them. All daily mean temperature layer that are temporally located during a temporary monthly map layer object are aggregated using the GRASS module *r.series*. The module *r.mapcalc* is then used to adjust the unit to degree Celsius. The monthly temporal extent of the temporary map layer object is copied to the resulting map layer. This map layer is then inserted into the temporal database and registered in the monthly STRDS. Finally the metadata of the new monthly STRDS is updated and printed to stdout.

```python
#!/usr/bin/env python
from datetime import datetime
from grass.pygrass import modules
import grass.temporal as tgis

tgis.init()

modules.Module("g.region", rast="temperature_daily.1")

temperature_daily = tgis.open_old_stds("temperature_daily", "strds")

temperature_monthly = tgis.open_new_stds(name="temperature_monthly",
                              type="strds",
                              temporaltype="absolute",
                              title="Monthly mean temperature Europe",
                              descr="Monthly mean temperature of Europe "
                                    "1950 - 1964 in degree Celsius",
                              semantic="mean",
                              overwrite=True)

dbif = tgis.SQLDatabaseInterfaceConnection()
dbif.connect()

granularity_list = []
start = datetime(1950, 1, 1)

for i in xrange(15 * 12):
    end = tgis.increment_datetime_by_string(start, "1 month")
    granule = tgis.RasterDataset(None)
    granule.set_absolute_time(start, end)
    granularity_list.append(granule)
    start = end

daily_list = temperature_daily.get_registered_maps_as_objects(dbif=dbif)

topo_builder = tgis.SpatioTemporalTopologyBuilder()
topo_builder.build(mapsA=granularity_list, mapsB=daily_list, spatial=None)

count = 0
for granule in granularity_list:
    count += 1
    if granule.contains:

        output_name = "%s_%i"%("temperature_monthly", count)

        aggregation_list = []
        for map_layer in granule.contains:
            aggregation_list.append(map_layer.get_name())

        modules.Module("r.series", input=aggregation_list,
                       output="result", method="average",
                       overwrite=True, quiet=True)

        modules.Module("r.mapcalc",
                       expression="%s = %s/%f"%(output_name, "result", 100.0),
                       overwrite=True, quiet=True)

        new_map_layer = tgis.RasterDataset("%s@%s"%(output_name,
                                           tgis.get_current_mapset()))
        new_map_layer.load()
        new_map_layer.set_temporal_extent(granule.get_temporal_extent())
```

```
        if new_map_layer.is_in_db(dbif=dbif) is False:
            new_map_layer.insert(dbif=dbif)
        else:
            new_map_layer.update(dbif=dbif)

        temperature_monthly.register_map(new_map_layer)

temperature_monthly.update_from_registered_maps(dbif=dbif)
temperature_monthly.print_info()

dbif.close()
```

Running the script will result in the following output:

```
"""
 +-------------------- Space-Time Raster Dataset -----------------------------+
 |                                                                            |
 +-------------------- Basic information -------------------------------------+
 | Id: ........................ temperature_monthly@soeren
 | Name: ...................... temperature_monthly
 | Mapset: .................... soeren
 | Creator: ................... soeren
 | Temporal type: ............. absolute
 | Creation time: ............. 2014-06-17 08:46:31.782002
 | Modification time:.......... 2014-06-17 08:47:30.440710
 | Semantic type:.............. mean
 +-------------------- Absolute time -----------------------------------------+
 | Start time:................. 1950-01-01 00:00:00
 | End time:................... 1965-01-01 00:00:00
 | Granularity:................ 1 month
 | Temporal type of maps:...... interval
 +-------------------- Spatial extent ----------------------------------------+
 | North:...................... 75.5
 | South:...................... 25.25
 | East:.. .................... 75.5
 | West:....................... -40.5
 | Top:........................ 0.0
 | Bottom:..................... 0.0
 +-------------------- Metadata information ----------------------------------+
 | Raster register table:......
     raster_map_register_2596306ec91749f8843adbdba48222dc
 | North-South resolution min:. 0.25
 | North-South resolution max:. 0.25
 | East-west resolution min:... 0.25
 | East-west resolution max:... 0.25
 | Minimum value min:.......... -32.162258
 | Minimum value max:.......... 5.229677
 | Maximum value min:.......... 12.125517
 | Maximum value max:.......... 33.13
 | Aggregation type:........... None
 | Number of registered maps:.. 180
 |
 | Title:
 | Monthly mean temperature Europe
 | Description:
 | Monthly mean temperature of Europe 1950 - 1964 in degree Celsius
 | Command history:
 | # 2014-06-17 08:46:31
 | ECAD_aggregation.py
 |
 +----------------------------------------------------------------------------+
"""
```

3.3 Sampling a raster time series at vector points

Listing 3: **Sampling European monthly mean temperatures with vector points.** The module *g.region* is used to set the correct spatial extent and resolution for sampling. The STRDS *temperature_monthly* is opened and a list of all registered raster layer objects is generated that is ordered by start time. The vector layer *observations* is opened which contains the coordinates of Paris, London and Berlin. The coordinates are read into a list. A raster layer will be opened for direct pixel access for each map layer object in the list. Each raster layer is sampled at specific positions for each vector point and the sampling result is printed to stdout.

```python
#!/usr/bin/env python
from datetime import datetime
from grass.pygrass.modules import Module
from grass.pygrass.raster import RasterRow
from grass.pygrass.vector import VectorTopo
from grass.pygrass.functions import coor2pixel
from grass.pygrass.gis.region import Region
import grass.temporal as tgis

tgis.init()

Module("g.region", raster="temperature_monthly.1")

temperature_monthly = tgis.open_old_stds("temperature_monthly", "strds")

monthly_list = temperature_monthly.get_registered_maps_as_objects(order="
    start_time")

region = Region()

capitals = ["Paris ", "London", "Berlin"]

vector_layer = VectorTopo("observations")
vector_layer.open("r")
vector_iterator = vector_layer.viter("points")

point_list = []

for point, capital in zip(vector_iterator, capitals):
    print capital, point
    point_list.append(point)

vector_layer.close()

for layer in monthly_list:
    start, end = layer.get_absolute_time()
    name = layer.get_name()

    raster_layer = RasterRow(name)
    raster_layer.open("r")

    for point, capital in zip(point_list, capitals):
        x, y = coor2pixel(point.coords(), region)
        value = raster_layer[int(x)][int(y)]
        print capital, start.date(), "-" ,end.date(), " value: ", value

    raster_layer.close()
```

Running the script will result in the following output:

```
"""
Paris  POINT(2.351667 48.856667)
London POINT(-0.118320 51.509390)
Berlin POINT(13.408056 52.518611)
Paris  1950-01-01 - 1950-02-01  value:  2.80483870968
London 1950-01-01 - 1950-02-01  value:  3.59225806452
Berlin 1950-01-01 - 1950-02-01  value:  -1.71967741935
Paris  1950-02-01 - 1950-03-01  value:  7.51857142857
London 1950-02-01 - 1950-03-01  value:  6.0625
Berlin 1950-02-01 - 1950-03-01  value:  2.82821428571
Paris  1950-03-01 - 1950-04-01  value:  8.62870967742
London 1950-03-01 - 1950-04-01  value:  7.65387096774
Berlin 1950-03-01 - 1950-04-01  value:  5.24225806452
Paris  1950-04-01 - 1950-05-01  value:  9.62633333333
London 1950-04-01 - 1950-05-01  value:  8.129
Berlin 1950-04-01 - 1950-05-01  value:  7.861
Paris  1950-05-01 - 1950-06-01  value:  15.2548387097
London 1950-05-01 - 1950-06-01  value:  11.6490322581
Berlin 1950-05-01 - 1950-06-01  value:  15.4722580645
...
Paris  1964-07-01 - 1964-08-01  value:  20.13
London 1964-07-01 - 1964-08-01  value:  17.5635483871
Berlin 1964-07-01 - 1964-08-01  value:  19.2822580645
Paris  1964-08-01 - 1964-09-01  value:  19.2403225806
London 1964-08-01 - 1964-09-01  value:  16.8190322581
Berlin 1964-08-01 - 1964-09-01  value:  16.6712903226
Paris  1964-09-01 - 1964-10-01  value:  17.6486666667
London 1964-09-01 - 1964-10-01  value:  15.099
```

```
Berlin 1964-09-01 - 1964-10-01  value:  14.168
Paris  1964-10-01 - 1964-11-01  value:  10.0380645161
London 1964-10-01 - 1964-11-01  value:  8.95903225806
Berlin 1964-10-01 - 1964-11-01  value:  7.99741935484
Paris  1964-11-01 - 1964-12-01  value:  7.721
London 1964-11-01 - 1964-12-01  value:  7.68933333333
Berlin 1964-11-01 - 1964-12-01  value:  5.38333333333
Paris  1964-12-01 - 1965-01-01  value:  3.32
London 1964-12-01 - 1965-01-01  value:  3.30580645161
Berlin 1964-12-01 - 1965-01-01  value:  1.46451612903
"""
```

3.4 Spatio-temporal intersection of trajectories

Listing 4: **Spatio-temporal intersection of animal tracking data from two Zebras in northern Botswana.** We need several helper functions to perform the intersection of the GPS tracking paths of two Zebras. The Zebra with the identifier 37743 has about 10900 GPS coordinates measured from 2007-10-28 to 2008-08-28. The Zebra with id 3864 has about 2600 GPS coordinates measured from 2007-10-29 to 2008-01-05. The function *read_csv_file()* will read the prepared CSV file of a single Zebra and creates based on the UTM-34S coordinates and the time stamp new *VectorDataset* objects managed in a list. The temporal extent is the event of measurement and the spatial extent is a single GPS point. To perform a spatio-temporal intersection we need to spatially buffer the GPS points. The function *spatial_buffer()* was designed to buffer the spatial extent of a *VectorDataset* using the distance to its nearest temporal neighbour in the future. The distance is divided by 8 and used for buffering. The function *temporal_topology_check()* checks for two map layer objects if they are temporally related and calls an intersection function when they are related. The intersection function *intersection()* performs the spatio-temporal intersection of the temporal and spatial extents of two *VectorDataset* objects. It computes the duration of the temporal intersection and the size of the areas resulting from the spatial intersection. This information is then printed to stdout. The code that makes use of these functions is located below the *intersection()* function. First the CSV files for each Zebra is read. Then the time stamps are snapped for each Zebra map layer object list, to create a valid temporal topology without gaps. The function for spatial buffering is called for each list and the spatio-temporal topology is built between the lists. Finally the spatial topology relations are checked and if a relation was found the temporal topology checker is called that performs the spatio-temporal intersection.

```python
#!/usr/bin/env python
from datetime import datetime
import csv
import math
from grass.pygrass import vector
from grass.pygrass.vector import geometry
from grass.pygrass import modules
import grass.temporal as tgis

def read_csv_file(filename):
    csv_reader = csv.reader(open(filename, "r"))
```

```
    layer_list = []
    for line in csv_reader:
        east=float(line[0])
        north=float(line[1])
        start = datetime.strptime(line[2], "%Y-%m-%d %H:%M:%S")

        tmp_layer = tgis.VectorDataset(None)
        tmp_layer.set_absolute_time(start, None)
        tmp_layer.set_spatial_extent_from_values(north=north, south=north,
                                                 east=east, west=east,
                                                 top=0, bottom=0)
        layer_list.append(tmp_layer)

    return layer_list

###############################################################################

def spatial_buffering(layer_list):
    for i in xrange(len(layer_list)):
        if i < (len(layer_list) - 1):
            ext_a = layer_list[i].spatial_extent.get_spatial_extent_as_tuple_2d()
            ext_b = layer_list[i + 1].spatial_extent.
                get_spatial_extent_as_tuple_2d()
            dx = ext_a[2] - ext_b[2]
            dy = ext_a[0] - ext_b[0]
            distance = math.sqrt(dx*dx + dy*dy)/8.0
        else:
            distance = 50

        layer_list[i].spatial_buffer_2d(distance)
        layer_list[i].buffer_size = distance

###############################################################################

def temporal_topology_check(layer, sublayer, srelation):
    if layer.overlaps and sublayer in layer.overlaps:
        intersection(layer, sublayer, srelation, "overlaps")
    if layer.overlapped and sublayer in layer.overlapped:
        intersection(layer, sublayer, srelation, "overlapped")
    if layer.equal and sublayer in layer.equal:
        intersection(layer, sublayer, srelation, "equal")
    if layer.during and sublayer in layer.during:
        intersection(layer, sublayer, srelation, "during")
    if layer.contains and sublayer in layer.contains:
        intersection(layer, sublayer, srelation, "contains")

###############################################################################

def intersection(layer_a, layer_b, srelation, trelation):
    spatial_extent = layer_a.spatial_intersection(layer_b)
    temporal_extent = layer_a.temporal_intersection(layer_b)

    extent = spatial_extent.get_spatial_extent_as_tuple_2d()
    duration = temporal_extent.get_end_time() - temporal_extent.get_start_time()

    if duration.days == 0 and duration.seconds < 60:
        return
    if layer_a.buffer_size > 1000 or layer_b.buffer_size > 1000:
        return

    area = spatial_extent.get_area()
    full_area = layer_a.spatial_extent.get_area() +\
                layer_b.spatial_extent.get_area()

    percent = 100*area/full_area

    print "Zebra 3864 met Zebra 3743 from", temporal_extent.get_start_time(),\
          "to", temporal_extent.get_end_time()
    print "    Duration:", duration, "\n    Area:",\
          int(area), "[m^2] about %i%% of full area,"%(percent)
    print "    Spatio-temporal relation:", srelation, trelation

###############################################################################
tgis.init()

zebra_3864_list = read_csv_file("Zebra_Id3864.csv")
zebra_3743_list = read_csv_file("Zebra_Id3743.csv")

tgis.AbstractSpaceTimeDataset.snap_map_list(zebra_3864_list)
tgis.AbstractSpaceTimeDataset.snap_map_list(zebra_3743_list)

spatial_buffering(zebra_3864_list)
spatial_buffering(zebra_3743_list)

```

```python
topo_builder = tgis.SpatioTemporalTopologyBuilder()
topo_builder.build(mapsA=zebra_3864_list, mapsB=zebra_3743_list, spatial="2D")

for layer in zebra_3864_list:
    if layer.overlap:
        for sublayer in layer.overlap:
            temporal_topology_check(layer, sublayer, "overlap")
    if layer.cover:
        for sublayer in layer.cover:
            temporal_topology_check(layer, sublayer, "cover")
    if layer.covered:
        for sublayer in layer.covered:
            temporal_topology_check(layer, sublayer, "covered")
    if layer.contain:
        for sublayer in layer.contain:
            temporal_topology_check(layer, sublayer, "contain")
    if layer.in_:
        for sublayer in layer.in_:
            temporal_topology_check(layer, sublayer, "in")
```

Running the script will result in the following output:

```
"""
Zebra 3864 met Zebra 3743 from 2007-12-27 23:00:53 to 2007-12-28 00:00:09
    Duration: 0:59:16
    Area: 1491 [m^2] about 8% of full area,
    Spatio-temporal relation: overlap during
Zebra 3864 met Zebra 3743 from 2007-12-28 00:00:13 to 2007-12-28 01:00:13
    Duration: 1:00:00
    Area: 1351 [m^2] about 15% of full area,
    Spatio-temporal relation: overlap overlaps
Zebra 3864 met Zebra 3743 from 2007-12-28 01:00:24 to 2007-12-28 02:00:43
    Duration: 1:00:19
    Area: 11708 [m^2] about 19% of full area,
    Spatio-temporal relation: contain contains
"""
```

3.5 Benchmark Python code

Listing 5: This Python code was used to perform the registration, selection and topology building benchmark.

```python
#!/usr/bin/env python
import grass.temporal as tgis
import datetime
import sys

def main(filename, strds, number_of_maps):
    # Initiate the temporal database, global variables and processes
    tgis.init()
    # A dedicated database connection object
    # speeds up the database access significantly
    dbif = tgis.SQLDatabaseInterfaceConnection()
    dbif.connect()

    # Start map registration in a space time dataset
    start = datetime.datetime.now()
    # Create a new space-time raster dataset (STRDS)
    new_strds = tgis.open_new_stds(name=strds,
                                   type="strds",
                                   temporaltype="absolute",
                                   title="Benchmark STRDS with %i map layers"%(
                                       number_of_maps),
                                   descr="Benchmark STRDS with %i map layers"%(
                                       number_of_maps),
                                   semantic="mean",
                                   overwrite=True, dbif=dbif)
    # Register raster layers in the STRDS
    tgis.register_maps_in_space_time_dataset(type="raster", name=strds,
                                             file=filename, start="2000-01-01",
                                             increment="1 second", interval=True,
                                             dbif=dbif, update_cmd_list=True)

    new_strds.update_from_registered_maps(dbif=dbif)

    end = datetime.datetime.now()
    print("Time for registration in [s]: ", end - start)
```

```python

    # Start SQL based selection operation
    start = datetime.datetime.now()

    sql_ret = new_strds.get_registered_maps(where="start_time >= '2000-01-01 00:00:00'")
    print("Length of returned dict", len(sql_ret))

    end = datetime.datetime.now()
    print("Time for map selection in [s]: ", end - start)

    # Start SQL based selection operation and generate a list of map objects
    start = datetime.datetime.now()

    map_list = new_strds.get_registered_maps_as_objects(where="start_time >= '2000-01-01 00:00:00'")
    print("Length of map list", len(map_list))

    end = datetime.datetime.now()
    print("Time for map selection and object generation in [s]: ", end - start)

    # Spatio-temporal topology between map layers
    start = datetime.datetime.now()
    # We reuse the map list to count the inner temporal relations
    relations = new_strds.count_temporal_relations(map_list, dbif=dbif)
    # We expect only follows and precedes relations
    print("Follows", relations["follows"])
    print("Precedes", relations["precedes"])

    end = datetime.datetime.now()
    print("Time for build st-topology and count relations in [s]: ", end - start)

    dbif.close()

if __name__ == "__main__":

    # python benchmark_run.py maps_list.txt A 1000000
    filename = sys.argv[1]
    strds = sys.argv[2]
    number_of_maps = int(sys.argv[3])

    main(filename, strds, number_of_maps)
```

References

[Bartlam-Brooks and Harris(2014)] H.L.A. Bartlam-Brooks and S. Harris. In search of greener pastures—using satellite images to predict the effects of environmental change on zebra migration, June 2014. url: https://www.movebank.org/node/11921.

[Gebbert and Pebesma(2014)] Sören Gebbert and Edzer Pebesma. A temporal GIS for field based environmental modeling. *Environmental Modelling & Software*, 53(0):1–12, 2014. ISSN 1364-8152. doi: 10.1016/j.envsoft.2013.11.001. URL `http://www.sciencedirect.com/science/article/pii/S136481521300282X`.

[Haylock et al.(2008)Haylock, Hofstra, Tank, Klok, Jones, and New] M. R. Haylock, N. Hofstra, A. M. G. Klein Tank, E. J. Klok, P. D. Jones, and M. New. A European daily high-resolution gridded data set of surface temperature and precipitation for 1950 - 2006. *Journal of Geophysical Research*, 113(D20), October 2008. ISSN 0148-0227. doi: 10.1029/2008JD010201.