Climate Data Tutorial

Pratik Singh, Sumet Khumphairan and Stella Dafka (Heidelberg University)

2024-09-23

Contents

Prerequisite

The following software and packages are needed:

- **ncdf4** (R package): For working with NetCDF files
- **raster** (R package): For working with raster data
- **ggplot2** (R package): For plotting
- **dplyr** (R package): For data manipulation
- **lubridate** (R package): For working with dates
- **terra** (R package): For spatial data handling
- **leaflet** (R package): For interactive maps
- **reshape2** (R package): For reshaping data
- **sf** (R package): For simple features and handling spatial data
- **exactextractr** (R package): For exact extraction of raster values from polygons

Introduction

This document contains exercises for **Day 1** of the IWR Summer School on "Applied Modeling of Climate-Sensitive Infectious Diseases," held on September 23, 2024. The goal of the tutorial is to equip participants with the tools and knowledge to handle climate data and understand how this data interacts with infectious diseases. Climate conditions, such as temperature and precipitation, directly impact the spread and intensity of many diseases (e.g., malaria, dengue, and tick-borne illnesses). To study this relationship, it is essential to manage and analyze data that has both spatial (geographical locations) and temporal (time) dimensions.

Workflow Overview

Load Climate Data: The first step in the tutorial is to load climate data, typically stored in NetCDF or raster formats, using packages like ncdf4, raster, and terra.

Manipulate and Process Data: After loading the data, the next step is data manipulation using packages like dplyr and sf to filter, summarize, or join spatial and non-spatial datasets.

Visualize the Data: Once the data is in a suitable format, the tutorial will guide you to visualize the spatial and temporal patterns using ggplot2, tmap, or leaflet. This visualization will help you explore the relationship between climate variables (e.g., temperature) and disease incidence.

Create Maps: A critical part of the tutorial involves creating static or interactive maps to illustrate how climate variables change over time and space. This is essential for studying climate-sensitive infectious diseases since spatial variability in climate can drive patterns of disease spread.

Downloading ERA5 Land Climate Data

This tutorial guides you through the process of downloading and analyzing ERA5 climate data from the Copernicus Climate Data Store (CDS).

Step 1: ERA5-Land hourly data

First, we will learn how to download ERA5-Land hourly data from 1950 to present using the Copernicus Climate Data Store (CDS). This reanalysis data is useful for climate research, including studying climate-sensitive infectious diseases.

Step 1.1: Set up your Copernicus account

Before downloading any data, you need to create a free account on the Copernicus website.

1. Go to Copernicus Climate Data Store.
2. Click "Sign up" in the top-right corner and follow the instructions to create your account.
3. Once signed up, log in to access the data download services.

Step 1.2: Navigate to ERA5 monthly data

Once logged in, you'll need to access the ERA5-Land hourly data from 1950 to present dataset. This dataset contains a variety of climate variables such as temperature, wind speed, and precipitation.

1. Go to the ERA5-Land hourly data dataset by navigating to the following link https://cds-beta.climate.copernicus.eu/datasets/reanalysis-era5-land?tab=overview
2. Read the Overview section to familiarize yourself with the dataset description, structure, and the variables available. Understanding the data will help you make informed choices during the selection process.

Step 1.3: Start the data download process

Now, let's walk through the data download process. Follow these steps to download the data:

1. Click on Download: After reviewing the data, scroll down and click on the "Download" button to begin the data selection process.

Step 1.4: Select data download parameters

Now, you'll select the specific parameters for your data download:

1. Variables of Interest:

a) The ERA5 dataset includes various climate variables like temperature, precipitation, wind speed, etc.

b) Select one variable at a time to avoid large data requests, which can slow down the download process.

c) For example, select 2m temperature if you're interested in temperature data.

2. Time Period:

a) You can choose specific time intervals for your data.

b) Select the Year(s), Month(s), Day(s), and Hour(s) that match the period of interest. For large datasets, be sure to carefully select the time periods to avoid downloading excessive amounts of data.

c) Example: If you want data for January 1st, 2022, at 14:00 hours, select Year = 2022, Month = January, Day = 01, and Time = 14:00.

3. Geographic Area:

a) Define the area of interest by specifying the geographic boundaries.

b) For instance, if you are interested in a specific region, you need to provide the xmin, xmax, ymin, and ymax coordinates that enclose the area. This will ensure that the downloaded data only covers the region you need.

c) Example: If your study area is between latitudes -3.09 and -2.34 and longitudes 32.76 and 34.01, fill in the following values: xmin: 32.76 xmax: 34.01 ymin: -3.09 ymax: -2.34

4. Sub-region Extraction:

Make sure to select Sub-region extraction to limit the data to your specified region. This is important for reducing the size of the data, ensuring it only includes the area you need.

5. Data Format:

Select the data format. The recommended format for this type of data is NetCDF4 (Experimental), as it is a widely-used format in climate data analysis and supports multidimensional arrays like time and spatial data.

6. Download Format:

For a single file, you can choose Unarchived (not zipped) to get the file directly. If you're downloading multiple files, it may be more convenient to choose Zip.

Step 1.5: Submit and Download

1. After selecting all the parameters (product type, variable, time period, geographic area, etc.), submit the download form.
2. The CDS system will process your request, which may take a few minutes depending on the size of the data request. Once processed, you can download the data to your local machine.

Step 1.6: Verify the Download

Once the data is downloaded, you can check if it covers the expected area and time period.

1. If you're downloading for the first time, it is recommended to download a small demo dataset to verify that your selected area matches the shapefile (geographic boundary) and time period. This ensures you don't waste time downloading large datasets with incorrect specifications.

Example: Select Year = 2022, Month = January, Day = 01, and Time = 14:00. Download this small test file and check if the area matches your region.

Step 1.7: Saving the File

1. Rename and Save the file: Once the .nc (NetCDF) file is downloaded, rename it for easy reference.

Example: Save the file as demo.nc in a specified folder where you will organize your climate data files.

Step 2: ERA5 monthly averaged data

After learning how to download ERA5 hourly data, let's now briefly walk through the steps for downloading ERA5 monthly averaged data.

Step 2.1: Go to ERA5 Monthly Averaged Data:

Navigate to the https://cds-beta.climate.copernicus.eu/datasets/reanalysis-era5-land-monthly-means?tab=overview page on the Copernicus website.

Step 2.2: Variables:

Choose from the available monthly averaged variables (e.g., 2m temperature, precipitation).

Step 2.3: Time Period:

Select the Year(s) and Month(s) of interest. Since this is monthly data, you won't be selecting days or hours.

Step 2.4: Geographic Area:

Similar to the hourly data, define the bounding box using the xmin, xmax, ymin, and ymax values to cover your region of interest.

Step 2.5: Sub-region Extraction:

Enable Sub-region extraction for a focused download.

Step 2.6: Data Format:

Choose NetCDF4 format, just as you did for the hourly data.

Step 2.7: Download Format:

Choose Unarchived (single file) or Zip if you are downloading multiple files.

Step 2.8: Submit and Download:

Submit your download request, and after processing, download the file to your local system.

Step 2.9: Rename and Save:

Rename the file appropriately (e.g., monthly_data.nc) and save it in your data folder.

Install and Load Required Packages

After downloading the ERA5 data, ensure that the necessary R packages are installed and loaded. If not already installed, run the following code:

```
# Install necessary packages (uncomment to run)
#if ("rstudioapi" %in% installed.packages()) {
#  rstudioapi::restartSession()
#}

# After restart, run package installation again
#install.packages(c("foreach", "ecmwfr", "raster", "sp", "sf", "geodata",
#                   "tidyr", "dplyr", "ncdf4", "ggplot2", "lubridate",
#                   "terra", "leaflet", "reshape2", "exactextractr"))
```

Now, load the required packages:

```
library(ncdf4)       # For working with NetCDF files
library(raster)      # For working with raster data
library(ggplot2)     # For plotting
library(dplyr)       # For data manipulation
library(lubridate)   # For working with dates
library(terra)       # For spatial data handling
library(leaflet)     # For creating interactive maps
library(reshape2)    # For reshaping data
library(sf)          # For working with simple features and spatial data
library(exactextractr) # For exact extraction of raster values from polygons
library(ecmwfr) #to interface with the ECMWF
```

Reading, Extracting, and Understanding Climate Data from NetCDF Files

Step 1: Print file summary

Downloading big data files from copernicus website will consume a lot of time, for example - Downloading global daily data for one year can take anywhere between 3 to 4 hrs. Hence, we have already downloaded some datasets for further tasks. Save the attached files in a folder and set the path to that folder while loading the required data.

```
# Open NetCDF file using the actual path
temp_global <- nc_open("/Users/pratik/Desktop/Tutorial_summer_school/ERA5land_global_t2m_2023_daily_0.5

# Print the summary of the file to understand the variables and dimensions
print(temp_global)
```

```
## File /Users/pratik/Desktop/Tutorial_summer_school/ERA5land_global_t2m_2023_daily_0.5.nc (NC_FORMAT_N
##
##      1 variables (excluding dimension variables):
##         short t2m[lon,lat,time]   (Chunking: [720,360,1])
##             long_name: 2 metre temperature
##             units: degC
##             add_offset: -15.399236346376
##             scale_factor: 0.00187235095519028
##             _FillValue: -32767
##             missing_value: -32767
```

```
##
##      3 dimensions:
##         time  Size:365   *** is unlimited ***
##             standard_name: time
##             long_name: time
##             units: days since 2023-1-1 00:00:00
##             calendar: gregorian
##             axis: T
##         lon  Size:720
##             standard_name: longitude
##             long_name: longitude
##             units: degrees_east
##             axis: X
##         lat  Size:360
##             standard_name: latitude
##             long_name: latitude
##             units: degrees_north
##             axis: Y
##
##     4 global attributes:
##         CDI: Climate Data Interface version 1.9.9rc1 (https://mpimet.mpg.de/cdi)
##         Conventions: CF-1.6
##         history: Sun May 05 11:21:04 2024: cdo remapbil,grid_0.5.nc ERA5land_global_t2m_2023_daily_0
## Sun May 05 11:17:33 2024: cdo -setattribute,t2m@units=degC -settunits,days -settaxis,2023-01-01,00:0
## 2024-05-05 09:11:10 GMT by grib_to_netcdf-2.33.0: grib_to_netcdf -k 4 -o t2m_2023_daily_0.1.nc ERA5l
##         CDO: Climate Data Operators version 1.9.9rc1 (https://mpimet.mpg.de/cdo)
```

Step 2: Extract Temperature Data

We extract the global temperature data using the variable name "t2m" (which stands for 2-meter temperature). This function returns a 3D array representing daily temperature values for each grid point over the specified time. Columns in the matrix corresponds to latitudes and rows are the longitudes.

```
# extracting the temperature data (mean) using the variable name in summary
temperature_data <- ncvar_get(temp_global, "t2m")
#print(temperature_data)           # Un-comment and print it
```

Step 3: Extract Latitude and Longitude Information

To understand the spatial coverage of the data, we extract latitude and longitude variables from the file.

```
## extract the latitude and longitude information of netcdf file
lat<-ncvar_get(temp_global,"lat")  ## lat is the variable name in net cdf file
#print(lat)

lon<-ncvar_get(temp_global,"lon")  ## lon is the variable name in net cdf file
#print(lon)
```

Step 4: Extract and Understand Time Variables

```
# now get the time variable (it is named as 'time')
time_units <- ncatt_get(temp_global, "time", "units")$value
print(time_units)            # unit is days (daily data) and it starts from 1-1-2023
```

```
## [1] "days since 2023-1-1 00:00:00"
```

```
# now time variable
time_temp <- ncvar_get(temp_global, "time")

## actual time corresponding to each time variable value
# origin is basically the starting time of data (time_units) in netcdf file
timestamp = as_datetime(c(time_temp*60*60*24),origin="2023-01-01")

#print(timestamp)              ## try to print it


# suppose instead of daily data, if netcdf file contains hourly data
# then actual time stamp for time varable containing
# 365*24 values (hourly data for 365 days) can be obtained by
# if Hours since origin is given
#timestamp = as_datetime(c(time_temp*60*60),origin="2023-01-01")  ## origin from time_units
```

Step 5: Check Temporal Coverage by Month

```
# checking the months covered in complete time frame
times_mon = month(timestamp)
#print(times_mon)   # try to print it
```

Step 6: Close nc file

Close the netcdf files after extracting the variables to free the memory.

```
# Close the NetCDF file after extracting data to free up memory
nc_close(temp_global)
```

Visualizing and Modifying the Climate Data

Step 1: Loading and Inspecting the Raster Data

Here, first we load the NetCDF file as a raster stack using the rast() function from the terra package. The print(temp_rast) function outputs key information about the raster, such as its extent (spatial coverage), dimensions (number of rows, columns, and layers), resolution (grid size), coordinate reference system (CRS), and time dimension.

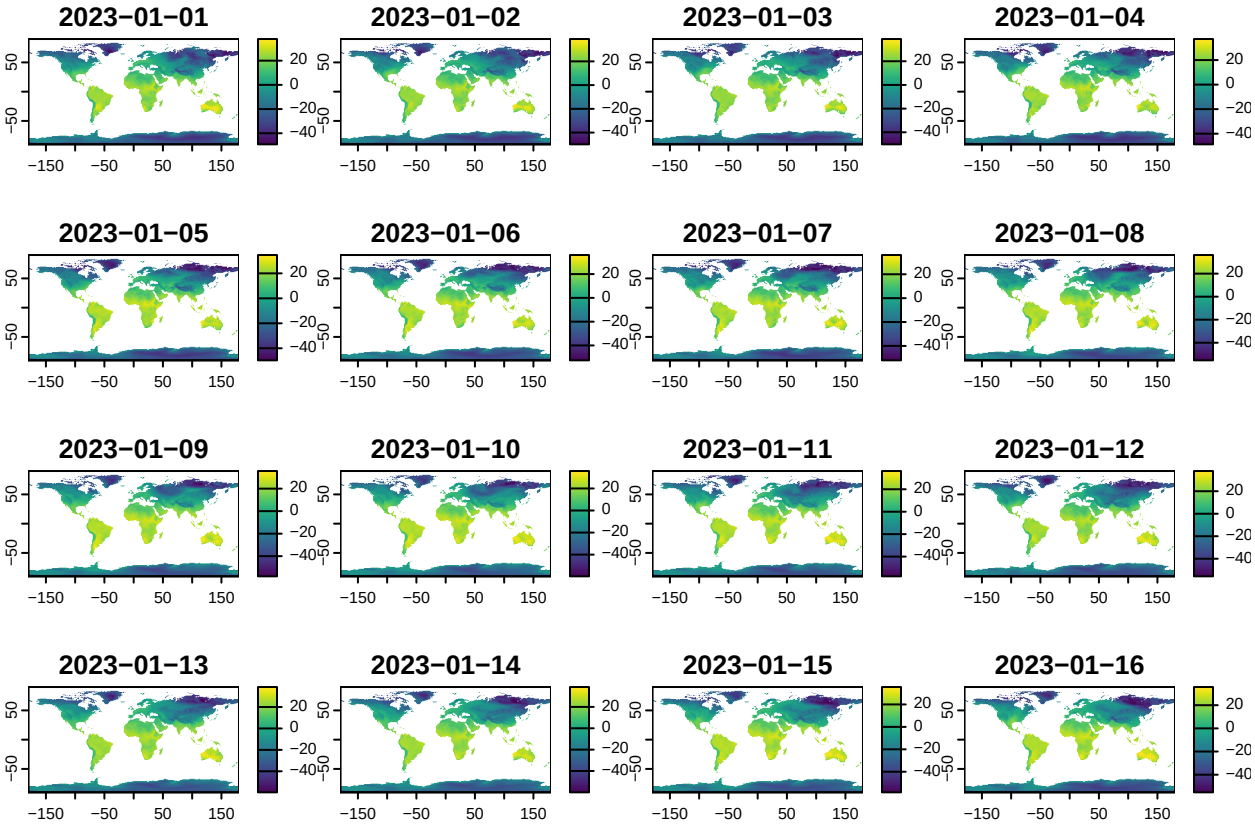
```r
# Load the raster data using the path
temp_rast <- rast("/Users/pratik/Desktop/Tutorial_summer_school/ERA5land_global_t2m_2023_daily_0.5.nc")
#print(temp_rast)

# Print information for a specific raster layer (e.g., 2nd Jan, 2023)
print(temp_rast[[2]])
```

```
## class       : SpatRaster
## dimensions  : 360, 720, 1  (nrow, ncol, nlyr)
## resolution  : 0.5, 0.5  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84
## source      : ERA5land_global_t2m_2023_daily_0.5.nc
## varname     : t2m (2 metre temperature)
## name        : t2m_2
## unit        :  degC
## time (days) : 2023-01-02
```

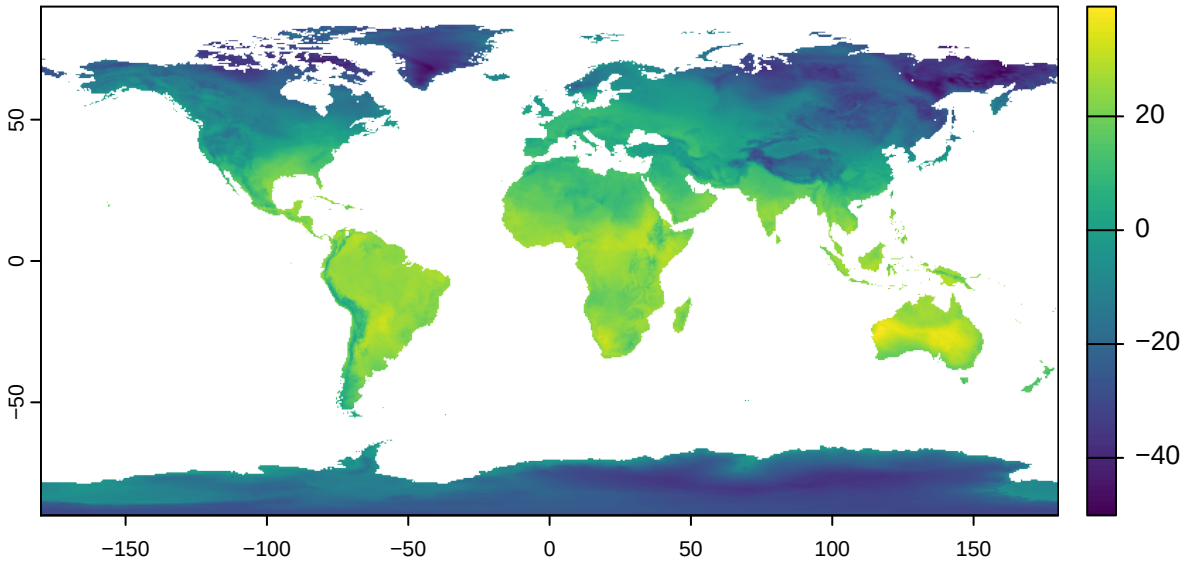
Step 2: Renaming Raster Layers with Timestamps

We rename the raster layers using timestamps to replace default names with actual dates. This makes the data more readable and helps in identifying specific days more easily when analyzing the dataset.

```r
# Change the layer names to actual dates using timestamp
names(temp_rast) <- timestamp
#print(temp_rast)

# Plotting the temperature data for all days
plot(temp_rast)
```




```
# Plotting data for a specific day (e.g., 2nd Jan, 2023)
plot(temp_rast[[2]])
```



Step 3: Understanding Coordinate Reference Systems (CRS)

A Coordinate Reference System (CRS) is essential when working with spatial data, particularly climate data. The CRS defines how spatial coordinates (like latitude and longitude) relate to real-world locations on the Earth's surface. Accurate geospatial analysis depends on a well-defined CRS to ensure that spatial data points are projected correctly, especially for large-scale datasets that span different regions or even the entire globe. Key Elements of a CRS

A CRS typically includes:

1. Datum: This is a mathematical model that defines the origin and orientation of the coordinate system relative to the Earth's shape (often represented as an ellipsoid).
2. Projection: This is the method used to flatten the 3D Earth surface onto a 2D map.
3. Units: The system can use either angular units like degrees (for latitude/longitude) or linear units like meters (for projected coordinates).

Common CRSs in Climate Data There are two widely used CRS models, WGS84 and NAD83. Each of these systems defines how to translate coordinates on a sphere (the Earth) into a 2D plane for use in mapping and analysis.

1. WGS84 (World Geodetic System 1984)

a) EPSG:4326 is the official code used to refer to this system.
b) WGS84 uses the Earth's center of mass as the origin and is widely used for GPS and global datasets.

c) It assumes a specific ellipsoid to model the Earth's shape.
d) Coordinates are typically expressed as latitude and longitude.
e) WGS84 is used globally, and it's the default system for most GPS devices.

2. NAD83 (North American Datum 1983)

a) EPSG:4269 is the official code for NAD83.
b) NAD83 is primarily used in North America and is based on a different ellipsoid than WGS84.
c) The center of NAD83's ellipsoid is not located at the Earth's center of mass, leading to slight differences from WGS84.
d) NAD83 differs from WGS84 by a few meters in North American coordinates due to the distinct origin and reference ellipsoid.

```
# Understanding the Coordinate Reference System (CRS)
crs_info <- crs(temp_rast)
print(crs_info)
```

```
## [1] "GEOGCRS[\"WGS 84\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOID[\"WGS 84\",63
```

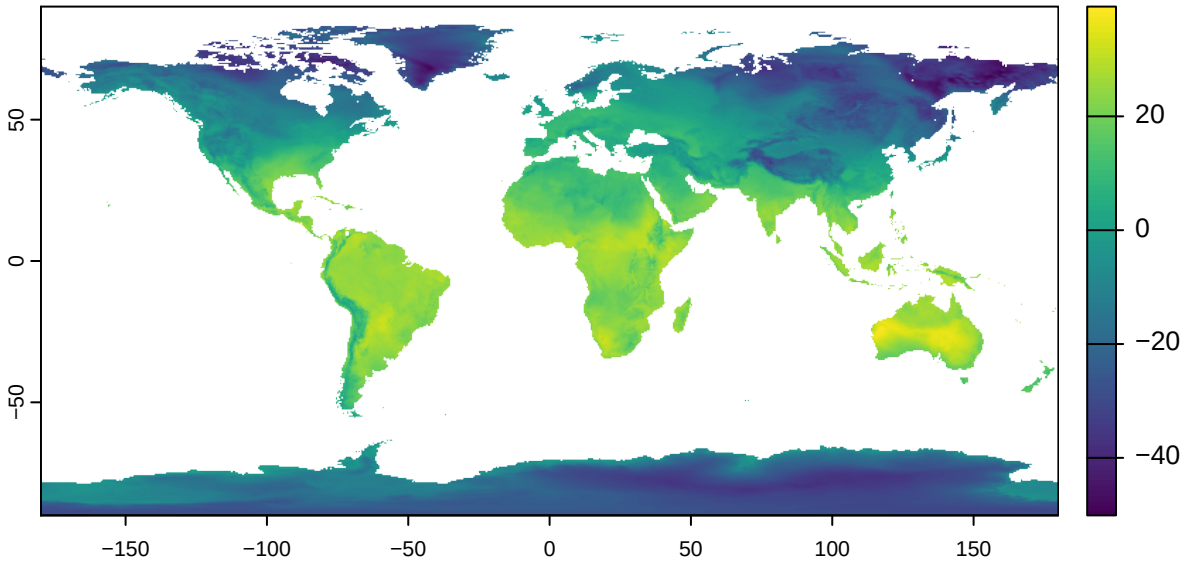
Step 4 Reprojecting to a Different CRS

Reprojecting the raster to a different CRS using project() helps align it with other spatial datasets. Although we reprojected to the same CRS in this example, reprojecting is useful when integrating data from different sources with different coordinate systems.

```
# Reprojecting to a different CRS (here we use the same CRS for demonstration)
#"EPSG:4326" is an identifier for a specific CRS known as WGS84 (World Geodetic System 1984)
temp_rast_reproj <- project(temp_rast, "EPSG:4326")
```

```
## |---------|---------|---------|---------|=========================================
```

```
#print(temp_rast_reproj)
plot(temp_rast_reproj[[2]])
```




```
# Reprojecting to crs NAD83 (to a different crd system)
temp_rast_nad83 <-  project(temp_rast, "EPSG:4269")
```

```
## |---------|---------|---------|---------|=========================================
```

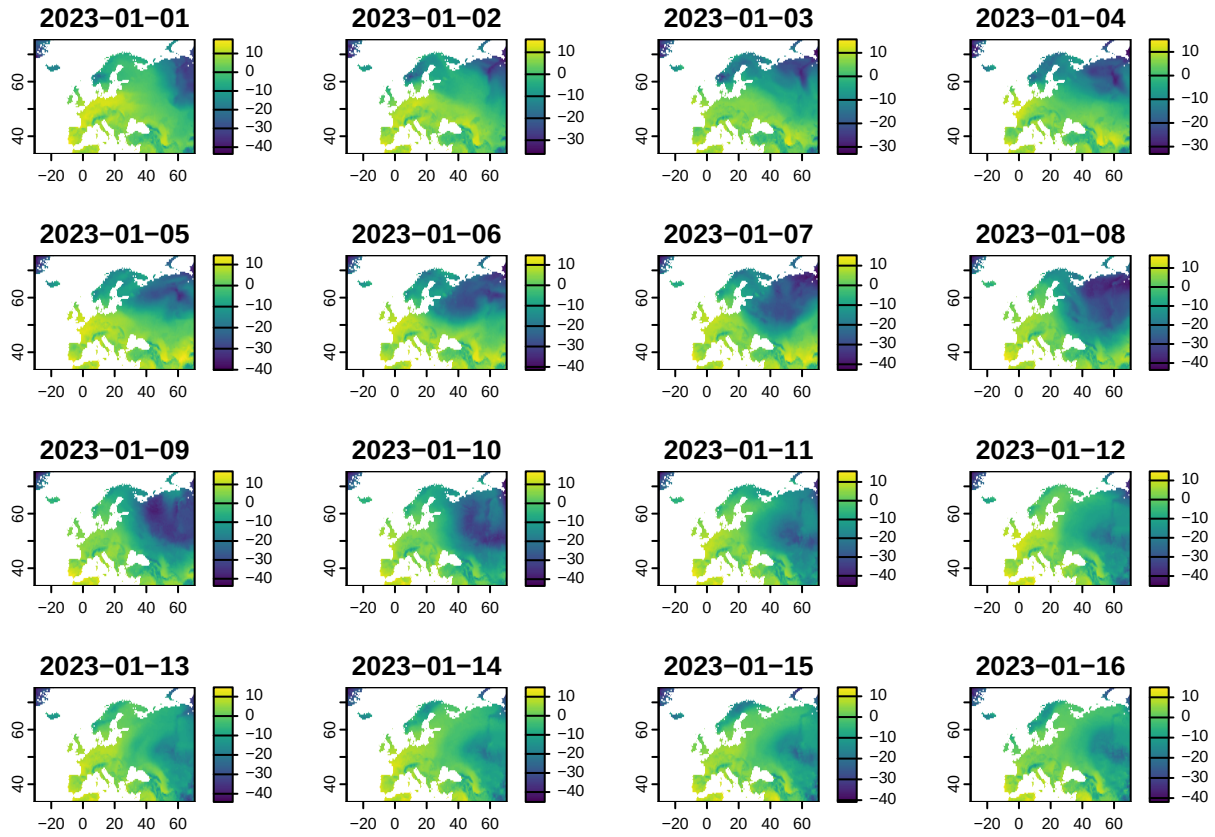
```
print(temp_rast_nad83)    # Now temp_rast_nad83 uses the NAD83 coordinate system (EPSG:4269)
```

```
## class       : SpatRaster
## dimensions  : 360, 720, 365  (nrow, ncol, nlyr)
## resolution  : 0.5, 0.5  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat NAD83 (EPSG:4269)
## source      : spat_1414d20cb37_321.tif
## names       : 2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, ...
## min values  :  -49.55841,  -50.10888,  -50.16879,  -50.42344,  -50.62003,  -52.52795, ...
## max values  :   38.18932,   39.28652,   38.84277,   37.57706,   35.42011,   36.16344, ...
## unit        :       degC,       degC,       degC,       degC,       degC,       degC, ...
## time (days) : 2023-01-01 to 2023-12-31
```

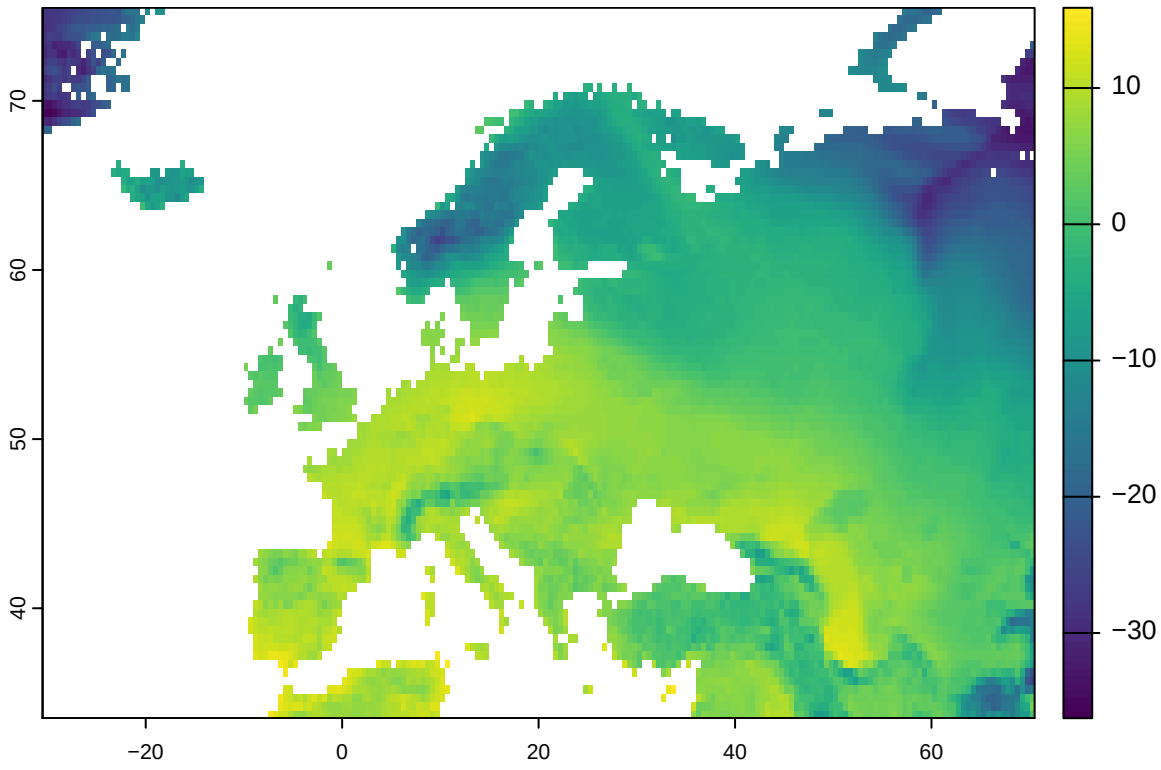
Step 5: Cropping the Raster to a Specific Extent

Suppose, we are only interested in temperature of Europe, so we will crop the global to extent of Europe for better visualization and it also helps us in saving memory while doing big computations. Extent of Europe (latitiude & longitude box) is around -40.5,75.5,25.5,75.5 (rough estimates).

```r
# Define the extent of Europe and crop the raster to this extent
## format to define extent is ((xmin, xmax, ymin, ymax) in raster plot)
europe_extent <- ext(-30.5, 70.5, 33.5, 75.5)
#It will crop the entire dataset for all days of year
europe_temp_rast <- crop(temp_rast, europe_extent)
#print(europe_temp_rast)
plot(europe_temp_rast)
```



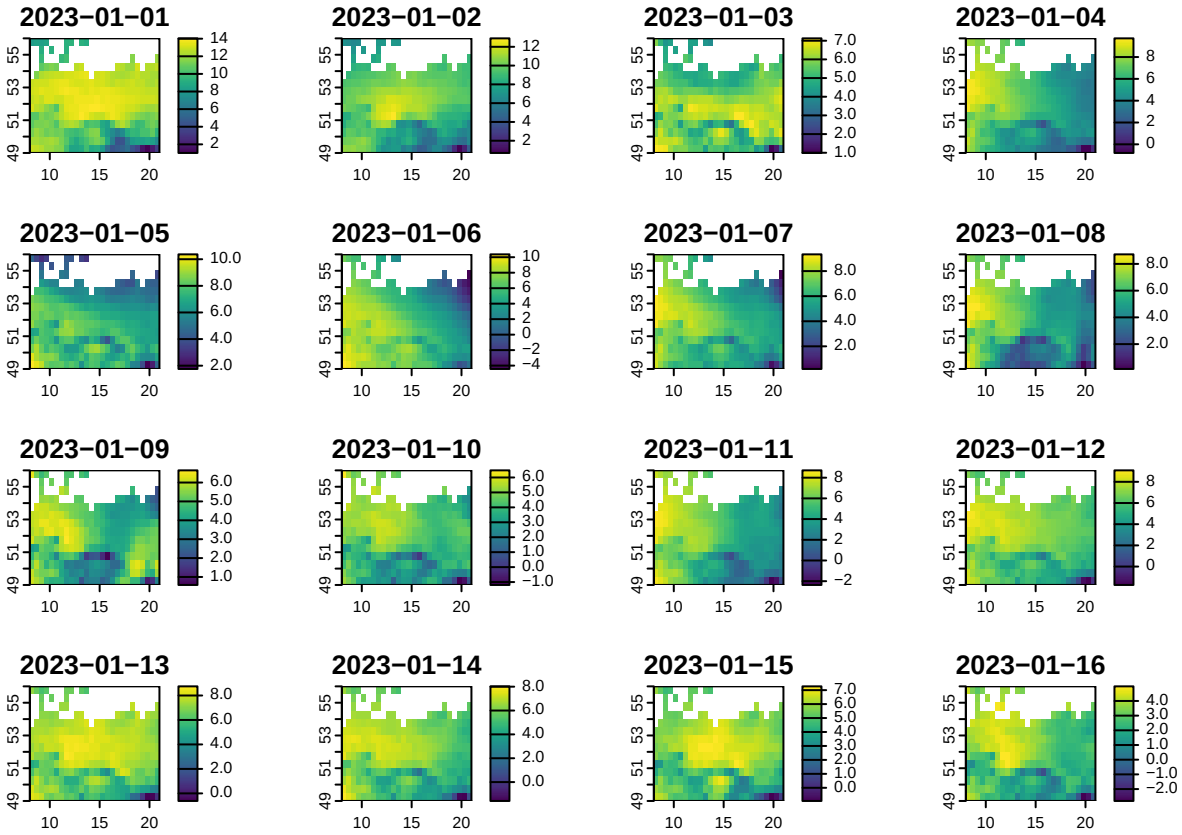

```r
plot(europe_temp_rast[[2]])
```



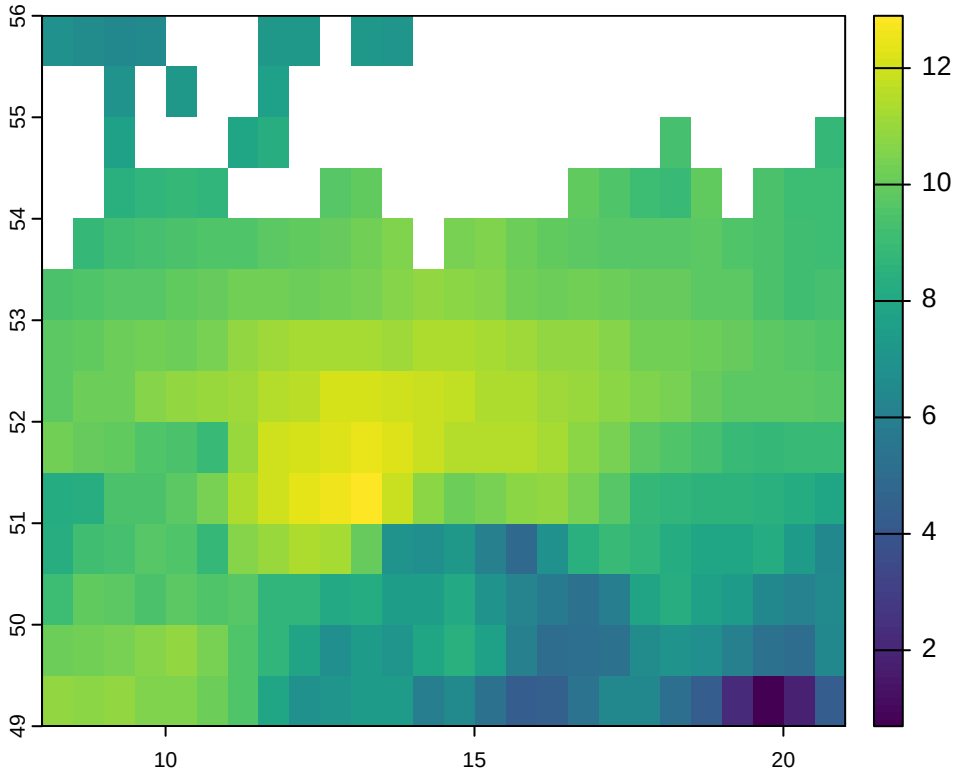
Step 6: Cropping to more finer extent

Let's further crop it to Heidelberg and then to the IWR building in which we are sitting right now:

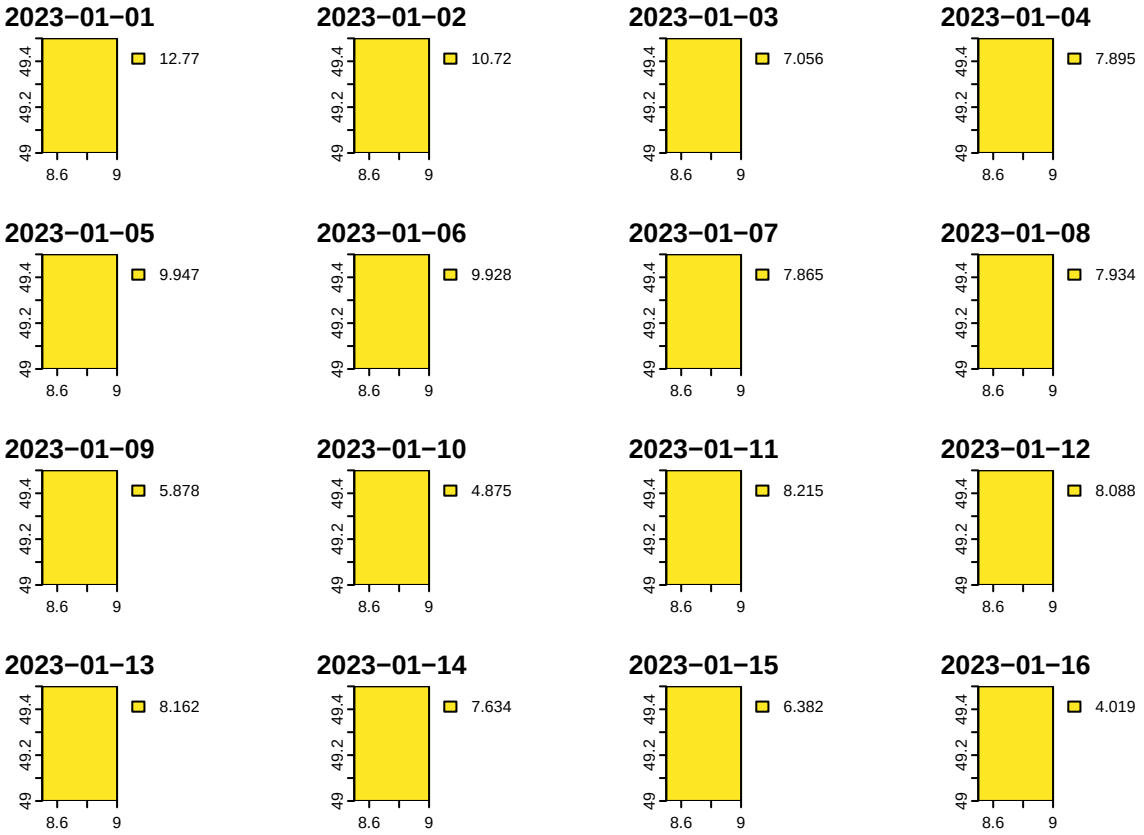
```
# Define the extent of Heidelberg and crop the raster
Heidelberg_extent <- ext(8, 21, 49, 56)
Heidelberg_temp_rast <- crop(temp_rast, Heidelberg_extent)
#print(Heidelberg_temp_rast)
plot(Heidelberg_temp_rast)
```




```
# mean temperature in Heidelberg (at 0.5 degree resolution approx 55*55 km^2)

plot(Heidelberg_temp_rast[[2]])
```




```
# Define a specific location (IWR) and crop the raster further
IWR_extent <- extent(8.5, 9, 49, 49.5)
IWR_temp_rast <- crop(Heidelberg_temp_rast, IWR_extent)
#print(IWR_temp_rast)  # Uncomment to see cropped extent details
plot(IWR_temp_rast)
```



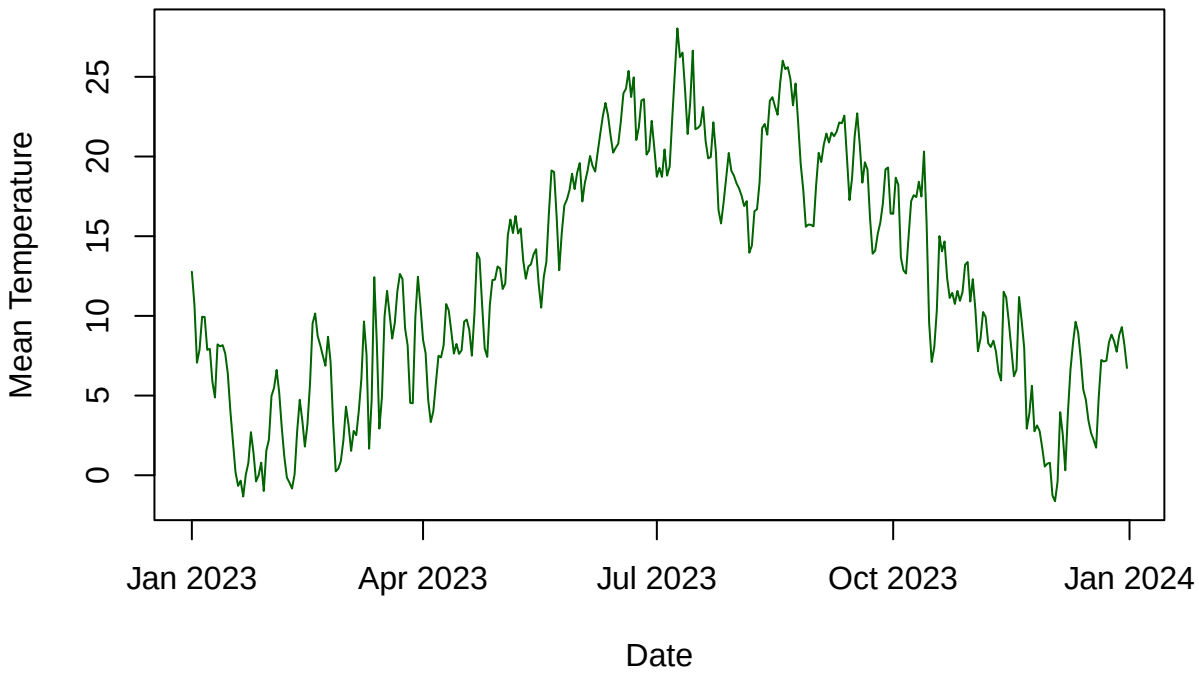
Bonus: We can also extract data using a single latitude - longitude value (it will basically extract the value of raster grid (extent) enclosing the point). For example - We take here specific coordinates for IWR as lat - 49.25 and lon - 8.75 (any point in between the raster grid of IWR extent defined above). Keep in mind the resolution of NETCDF file.

```
lat_IWR <- 49.25        # falls within latitude extent (49,49.5)
lon_IWR <- 8.75         # falls within longitude extent (8.5,9)
values_at_IWR <- extract(temp_rast, matrix(c(lon_IWR, lat_IWR), ncol=2))
#print(values_at_IWR)                ## same as IWR_temp_rast
```

Step 7: Plotting a Time Series of Temperature for IWR

For visualizing the yearly temperature throughout the year, lets look at the time series curve of complete year

```
# Plotting a time series for mean temperature for IWR
plot(timestamp, values_at_IWR, type = "l", xlab = "Date", ylab = "Mean Temperature", pch = 26, col = "da
```



Spatial Aggregation of Climate Data

Spatial aggregation involves modifying the resolution of climate data to combine or reduce the number of data points per unit area. This process can help manage large datasets and analyze climate data at different spatial scales.

Step 1: Concept

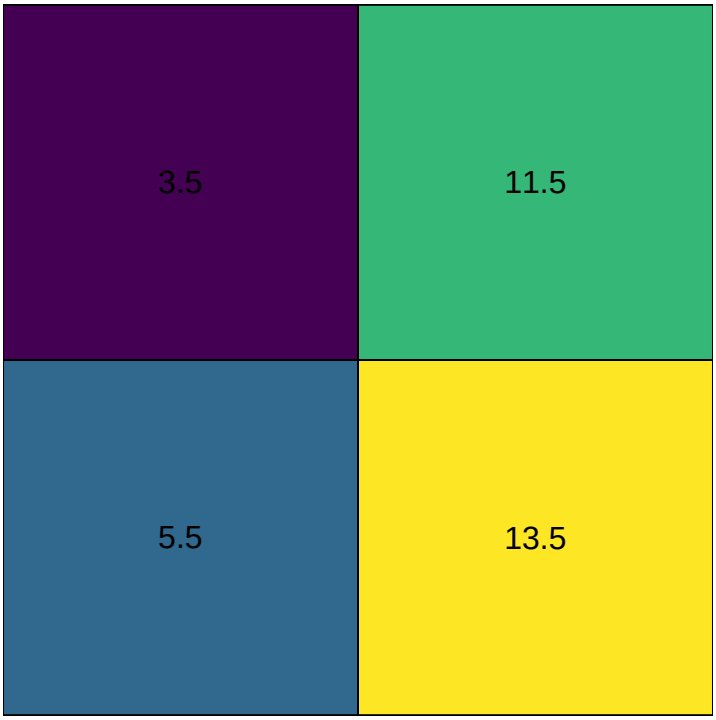
For conceptual understanding of above concept let's create a 4 x 4 matrix matrix and transform it into raster object so that it becomes 4 by 4 raster containing values between 1 and 16.

```
# Create a 4x4 matrix and convert it into a raster
m <- matrix(1:16, ncol = 4, nrow = 4)
r16 <- rast(m)
#print(r16)

# Function to visualize raster data
plot_raster <- function(r) {
  plot(r, axes = FALSE, legend = FALSE)
  plot(as.polygons(r, dissolve = FALSE, trunc = FALSE), add = TRUE)
  text(r, digits = 2)
}
```

```
# Plot the created raster matrix
plot_raster(r16)
```

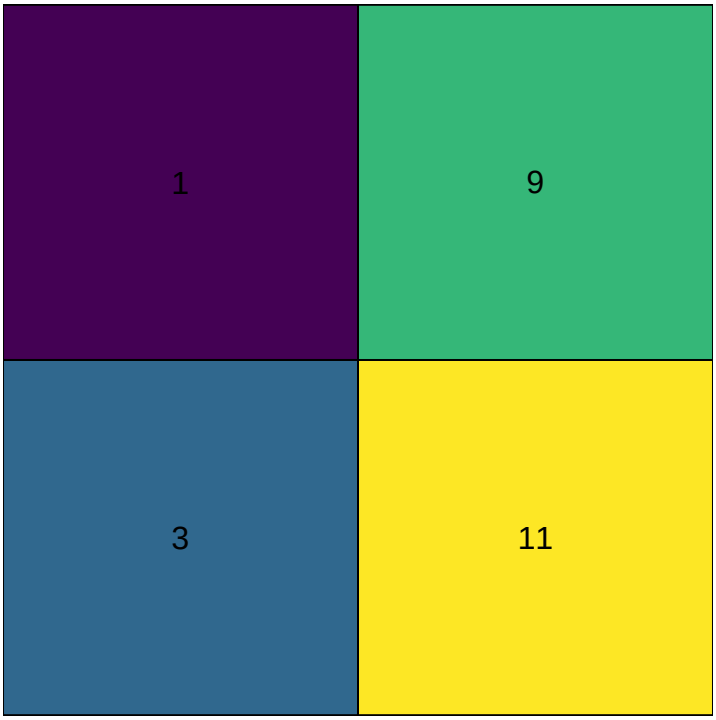
Here, we want to aggregate the raster r16 so it becomes a raster of 2 by 2 (i.e. 4 pixels) i.e. the raster will aggregate using 2 pixels within the horizontal and the vertical directions.

```
# Aggregate the raster to a 2x2 resolution using default aggregation function
r4 <- aggregate(r16, fact = 2)

#print(r4)  # Uncomment to check new resolution
plot_raster(r4)
```



By default, the aggregate() function is using the mean() function to summarize the pixel values. Hence, the upper left pixel has a value of 3.5 which correspond to the average of 4 pixels $(1 + 5 + 2 + 6) / 4 = 3.5$. But one can use any function (default is mean) that returns a single value such as the min() or max() function of the aggregating pixels.

```
# Aggregate using the minimum value instead of the mean
plot_raster(aggregate(r16, fun = "min"))
```



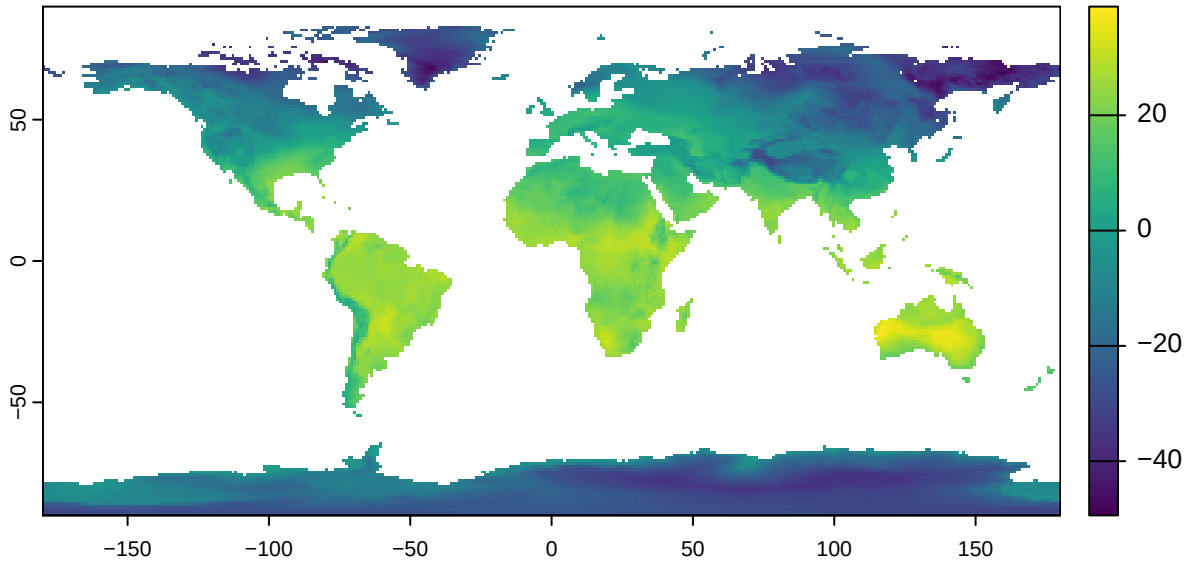
Step 2: Spatially aggregating climate data

Suppose, we want to Change the resolution of data (let's say we want to change it from 0.5-0.5 to 1-1 degree) of the original raster file of world (temp_rast), which is originally at 0.5*0.5 resolution

```
# Aggregating the raster to 1-degree resolution
temp_rast_1 <- aggregate(temp_rast, fact = 2)
print(temp_rast_1)  # Check new resolution (now 1-degree resolution i.e. approx 110*110 km^2)
```

```
## class       : SpatRaster
## dimensions  : 180, 360, 365  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84
## source(s)   : memory
## names       : 2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, ...
## min values  :  -48.71304,  -49.40253,  -49.63798,  -48.07597,  -49.60381,  -51.68540, ...
## max values  :   37.76008,   38.85120,   38.35502,   36.50561,   34.48394,   35.54931, ...
```

```
plot(temp_rast_1[[2]])
```

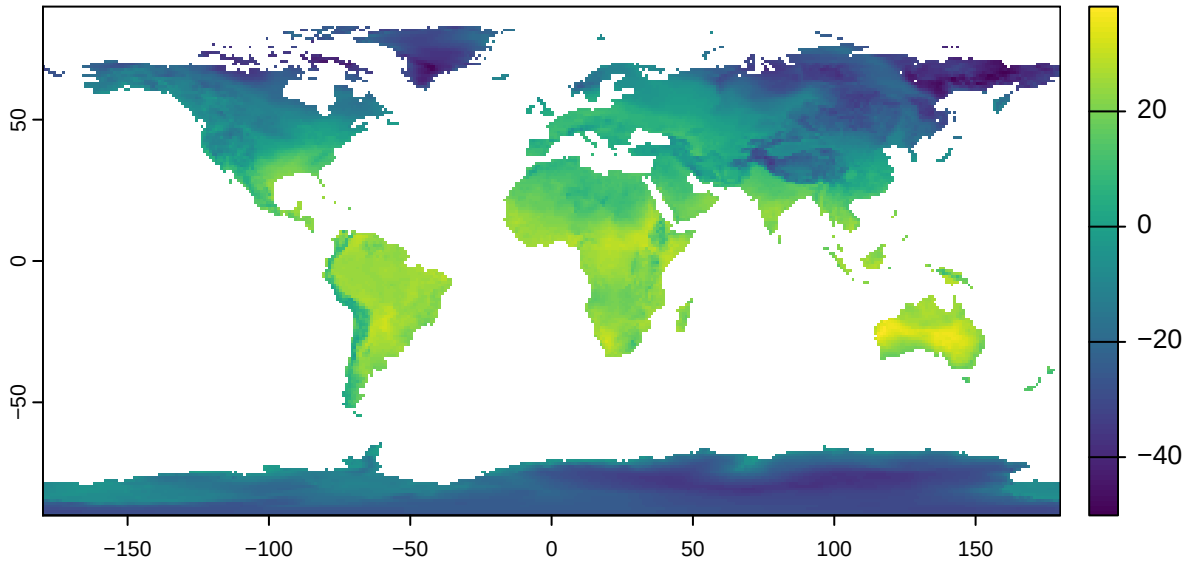


Note that default aggregation function is weighted mean (since area of each pixel is same), it means mean temperature of 4 (0.5-0.5) grids (2 along latitude and 2 along longitude) combined into 1 for 1*1 degree resolution.

```
#similarly aggregating our temp_rast to 1 degree resolution using minimun temperature of grid combined
temp_rast_1_min <- aggregate(temp_rast, fact = 2, fun = "min")
print(temp_rast_1_min)  ## check the values and compare with print(temp_rast_1) where it was aggregated
```

```
## class       : SpatRaster
## dimensions  : 180, 360, 365  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84
## source(s)   : memory
## names       : 2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, ...
## min values  :  -49.55841,  -50.10888,  -50.16879,  -50.42343,  -50.62003,  -52.52796, ...
## max values  :   37.46847,   38.18558,   37.75119,   35.86012,   34.08700,   34.92581, ...
```

```
plot(temp_rast_1_min[[2]])      # plot of 2nd layer
```

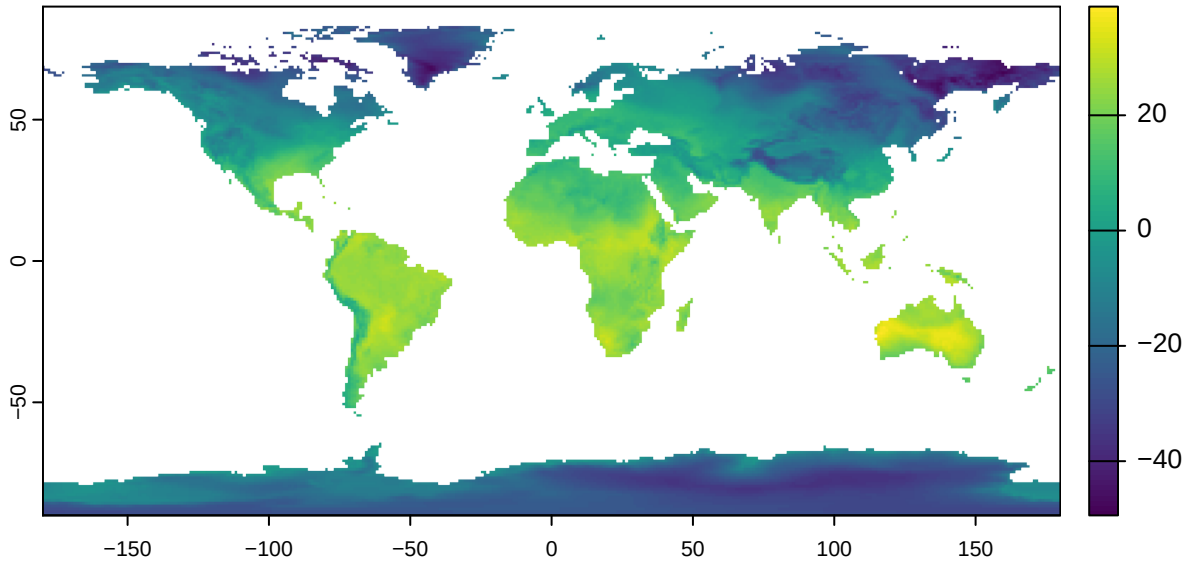


Step 3: Disaggregating Data to Original Resolution

```
# Disaggregate back to 0.5-degree resolution from 1-degree resolution
temp_rast_dis <- disagg(temp_rast_1, fact = 2)
```

```
## |---------|---------|---------|---------|==============
```

```
#print(temp_rast_dis)  # Uncomment to check the new resolution
plot(temp_rast_dis[[2]])
```



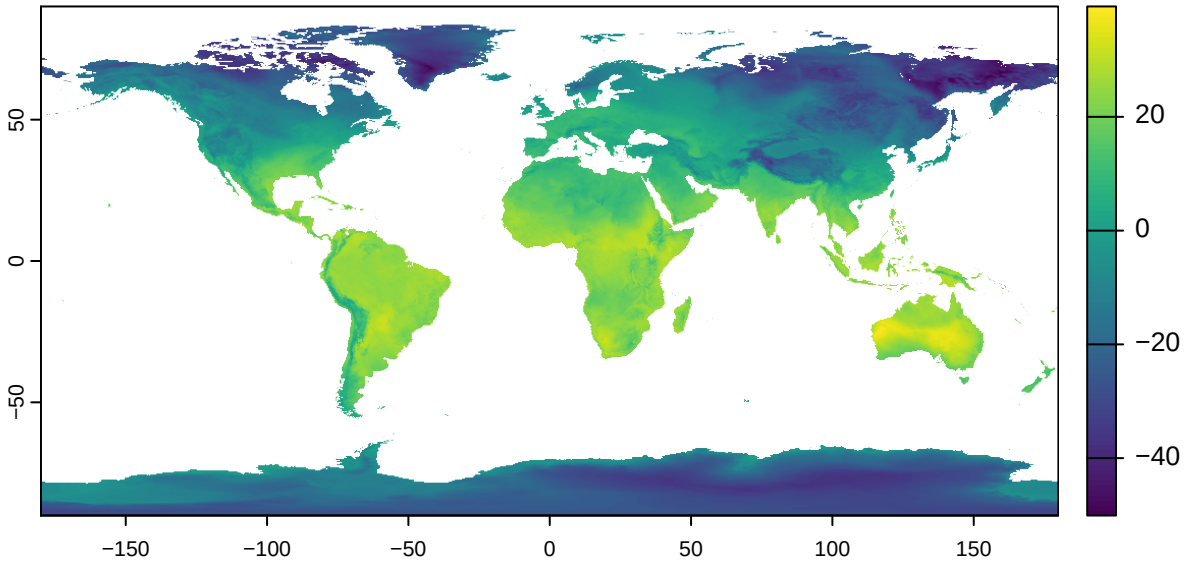
Step 4: Resampling Raster Data to Match Different Resolutions

We can also change the resolution of one raster file to the resolution of another raster that does not have the same geometry (i.e. cell size), the process is called re sampling. For example - suppose we have global daily mean temperature file as above but with resolution of 0.1 degree (unlike the one we are working with (0.5 degree resolution)).

```
temp_global_0.1_rast <- rast("/Users/pratik/Desktop/Tutorial_summer_school/ERA5land_global_t2m_2023_dai
print(temp_global_0.1_rast)    ## check the resolution (0.1 * 0.1)
```

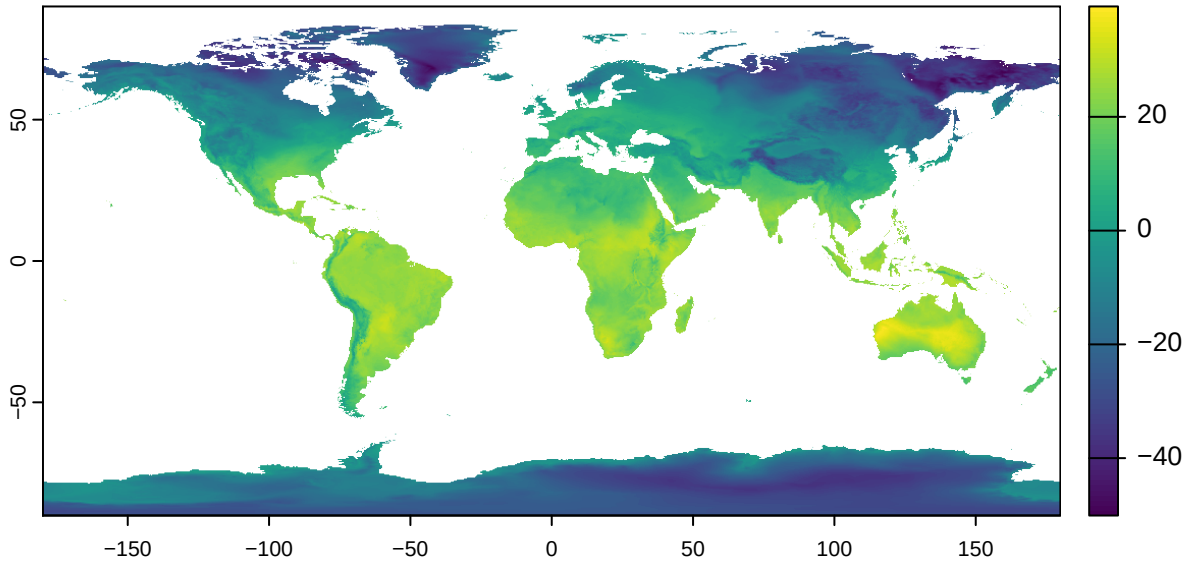
```
## class       : SpatRaster
## dimensions  : 1801, 3600, 365  (nrow, ncol, nlyr)
## resolution  : 0.1, 0.1  (x, y)
## extent      : -180.05, 179.95, -90.05, 90.05  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs
## source      : ERA5land_global_t2m_2023_daily_0.1.nc
## varname     : t2m (2 metre temperature)
## names       : t2m_1, t2m_2, t2m_3, t2m_4, t2m_5, t2m_6, ...
## unit        :  degC,  degC,  degC,  degC,  degC,  degC, ...
## time (days) : 2023-01-01 to 2023-12-31
```

```
plot(temp_global_0.1_rast[[2]])
```



Now we will use our working temp_rast(0.5 degree resolution) file to aggregate temp_global_0.1_rast to the resolution of 0.5. If we convert the entire raster layer (365 days) of temp_global_0.1_rast to 0.5 resolution, it will take a very long time, hence converting the resolution of only one layer of temp_global_0.1_rast. Note that we can still use the temp_rast file to resample one layer to 0.5 even though temp_rast is of 365 layer, as only the spatial resolution of temp_rast (0.5) is of importance. So we will use the complete temp_rast object (at 0.5 degree resolution) to convert one layer of raster object at 0.1 resolution.

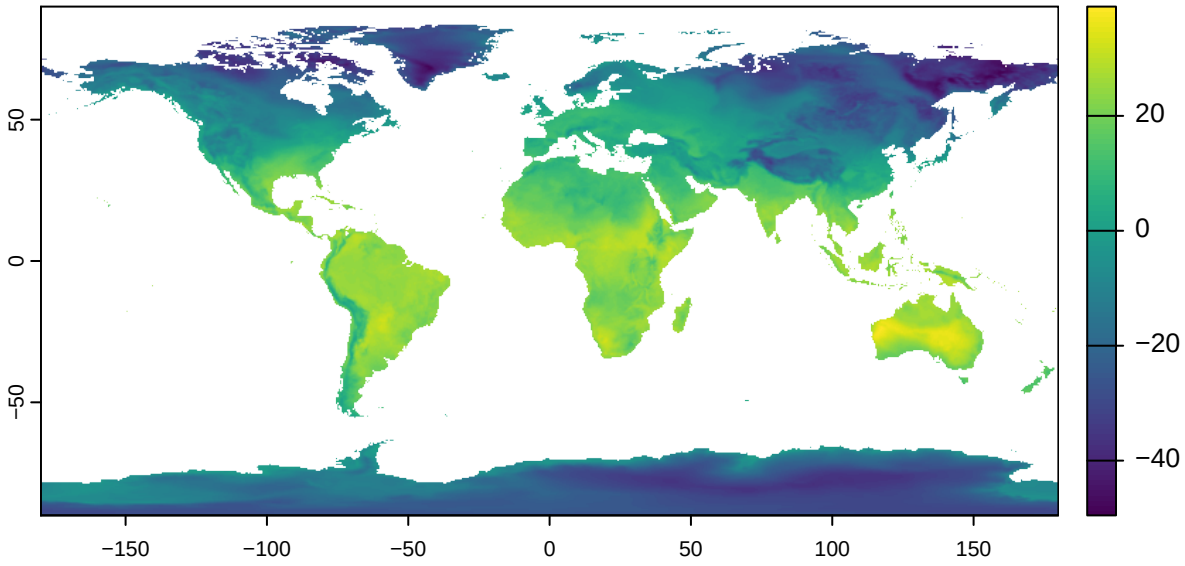
```
one_layer <- temp_global_0.1_rast[[2]]
plot(one_layer)                ## resolution of single layer
```




```
## now resampling one_layer to resolution temp_rast
temp_global_converted_rast <- resample(one_layer, temp_rast)
print(temp_global_converted_rast)      ## check the resolution
```

```
## class       : SpatRaster
## dimensions  : 360, 720, 1  (nrow, ncol, nlyr)
## resolution  : 0.5, 0.5  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs
## source(s)   : memory
## name        :     t2m_2
## min value   : -49.53293
## max value   :  39.02011
## unit        :      degC
## time (days) : 2023-01-02
```

```
plot(temp_global_converted_rast)   ## plot the aggregated 0.5 raster layer
```



Temporal aggregation of Climate Data

Temporal aggregation simplifies climate data analysis by summarizing high-frequency data (e.g., hourly or daily) over longer periods such as months, seasons, or years. This process helps to reduce short-term variability and smooth out daily fluctuations, allowing for the identification of more meaningful, long-term climate trends. It also makes it easier to handle large datasets by reducing their size, while retaining key insights. Temporal aggregation is particularly valuable in studying climate change, where long-term trends such as warming or shifts in precipitation patterns are of primary interest. This approach is critical for uncovering subtle, but important, changes in climate behavior over extended periods.

Step 1: Monthly Aggregation of daily climate Data

Since, our temp_rast file contains daily data we will aggregate it to monthly temperature values for all regions (Monthly aggregation stands for mean over all the days of month).

```
# Extract date information from the layer names
dates <- as.Date(names(temp_rast))  ## as done earlier in Chapter 2
#print(dates)

# Create a vector representing the months
months <- format(dates, "%Y-%m")
#print(months)
```

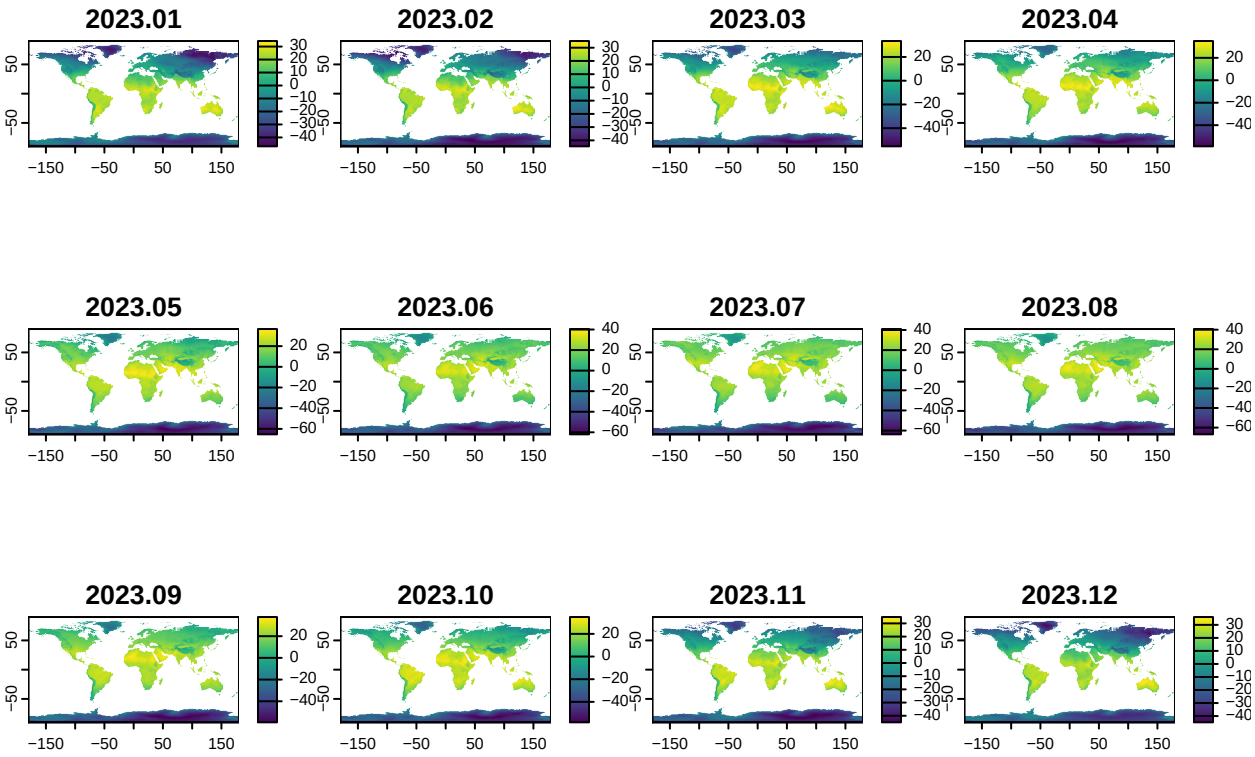
```r
# Aggregate daily data to monthly means using the 'tapp' function
#tapp function (from terra package) aggregates across layers based on the grouping factor
#Here, 'months' is used as groupin factor
monthly_temp <- tapp(temp_rast, months, fun = mean)

# Check the resulting SpatRaster with monthly data
print(monthly_temp)  ## check the temporal layears (3D matrix layer)
```

```
## class       : SpatRaster
## dimensions  : 360, 720, 12  (nrow, ncol, nlyr)
## resolution  : 0.5, 0.5  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84
## source(s)   : memory
## names       :  X2023.01,  X2023.02,  X2023.03,  X2023.04,  X2023.05,  X2023.06, ...
## min values  : -46.91308, -44.62891, -55.42388, -58.86212, -65.26302, -62.27098, ...
## max values  :  34.28631,  35.05594,  33.64621,  35.35714,  36.97422,  40.96340, ...
```

```r
# Get and clean the current layer names
current_names <- names(monthly_temp)
# Remove the "X" prefix
#X prefix got introduced due to conversion from numeric format of names to character format
cleaned_names <- sub("^X", "", current_names)

# Assign the new names to the SpatRaster object
names(monthly_temp) <- cleaned_names

# Check the new names after cleaning
print(monthly_temp)
```

```
## class       : SpatRaster
## dimensions  : 360, 720, 12  (nrow, ncol, nlyr)
## resolution  : 0.5, 0.5  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84
## source(s)   : memory
## names       :   2023.01,   2023.02,   2023.03,   2023.04,   2023.05,   2023.06, ...
## min values  : -46.91308, -44.62891, -55.42388, -58.86212, -65.26302, -62.27098, ...
## max values  :  34.28631,  35.05594,  33.64621,  35.35714,  36.97422,  40.96340, ...
```

```r
# Plot the monthly aggregated data
plot(monthly_temp)
```



Step 2: Seasonal Aggregation of daily climate Data (try as an exercise without looking at solution)

Problem - Suppose we want to aggregate the annual data season wise (i.e. summer, winter , autumn and spring each containing the mean over daily data for 3 months).

In general, seasons are defined as follows:

1. Winter: December to February (12, 1, 2) (For convenience we will take the 12th month of same year, although practically we take the data of 12th month of previous year for defining winter)
2. Spring: March to May (3, 4, 5)
3. Summer: June to August (6, 7, 8)
4. Autumn: September to November (9, 10, 11)

Using the definition of seasons given above, try to aggregate the data and plot the aggregated raster for each seasons?

Hint - Create a vector representing the seasons based on the dates (search for for months in date format - Recall chapter 2 (date and origin))

```
# Create a vector representing the seasons based on the dates
seasons <- ifelse(format(dates, "%m") %in% c("12", "01", "02"), "Winter",
                  ifelse(format(dates, "%m") %in% c("03", "04", "05"), "Spring",
```

```
                          ifelse(format(dates, "%m") %in% c("06", "07", "08"), "Summer",
                                 "Autumn")))
```

Extracting Data and Converting into Working Data Frames

This section covers how to prepare climate data for data-driven models by extracting relevant information, converting it into a usable format like data frames, and visualizing it. The process involves handling grid-level data, transforming it into a format suitable for model input, and adapting it for use with regional shapefiles (next section). The goal is to create working data frames that can be directly applied in statistical and mathematical models. This preparation is essential for aligning the data with geographic boundaries and ensuring compatibility for subsequent regional analysis.

Step 1: Reviewing and Structuring Temperature Data

We will again call our variable "temperature_data", lat and lon from chapter 2, which contains the mean temperature, latitude and longitude , let's print those values to refresh the information stored

```
#print(temperature_data)      ## temperature data of 365 layers
```

Here, as mentioned in Chapter 2, each layer of data has latitude (columns) and longitude (row), and each data point in matrix has exact position (columns heading represent latitude and row heading represent longitude) and 365 layers for 365 days. Let's look at the latitudes and longitudes once again-

```
# Uncomment if needed to verify latitude and longitude assignment
#print(lat)             ## latitudes at 0.5 degree resolution
#print(lon)             ## longitudes at 0.5 degree resolutions

# Assign row names as longitude and column names as latitude to the temperature data matrix
colnames(temperature_data) <- lat
rownames(temperature_data) <- lon

# Create a data frame of all coordinates
temperature_data_df <- melt(temperature_data)
#print(temperature_data_df)
```

In dataframe temperature_data_df, var1 is longitude, var2 is latitude var 3 is time of year (from 1 to 365) and values are the mean temperature corresponding to latitude, longitude and time. Since var3 is time and it ranges from 1 to 365 days, and it will take lot of time to work with all days simultaneously so, from here to save time and quickly show the results, we will work with only data points where Var3 = 2 i.e. 2nd January, 2023 (In terms of matrix, it will have now one layer of 2nd day of year 2023 for all latitude and longitudes)

```
# Filter rows where Var3 (time) equals 2 (2nd Jan, 2023)
temp_data_df2 <- temperature_data_df %>%
  filter(Var3 == 2)

# Remove the Var3 column (we don't need the time column now as we know that it is the data of 2nd Jan, 
temp_data_df2 <- temp_data_df2 %>%
  select(-Var3)

# Print the resulting data frame to inspect
#print(temp_data_df2)     ### uncomment to print
```

Now we can save this data frame as csv file with proper format to utilize the data in data driven models (along with other variables like precipitation, population , GDP etc). But often the response variable (or the surveillance data which we are trying to model with these climate covariates) is not at grid level but on a more standard spatial level. In Europe most of these data are collected at NUTS level (Nomenclature of Territorial Units for Statistics). It is a hierarchical system developed by Eurostat, the statistical office of the European Union developed to divide the economic territory of the European Union and its candidate countries into regions for the purposes of statistical analysis. In Brazil, these surveillance data are collected at municipality level and so on in other countries so we use the shapefiles developed by these organization to extract the climate data of these regions instead of working at grid level.

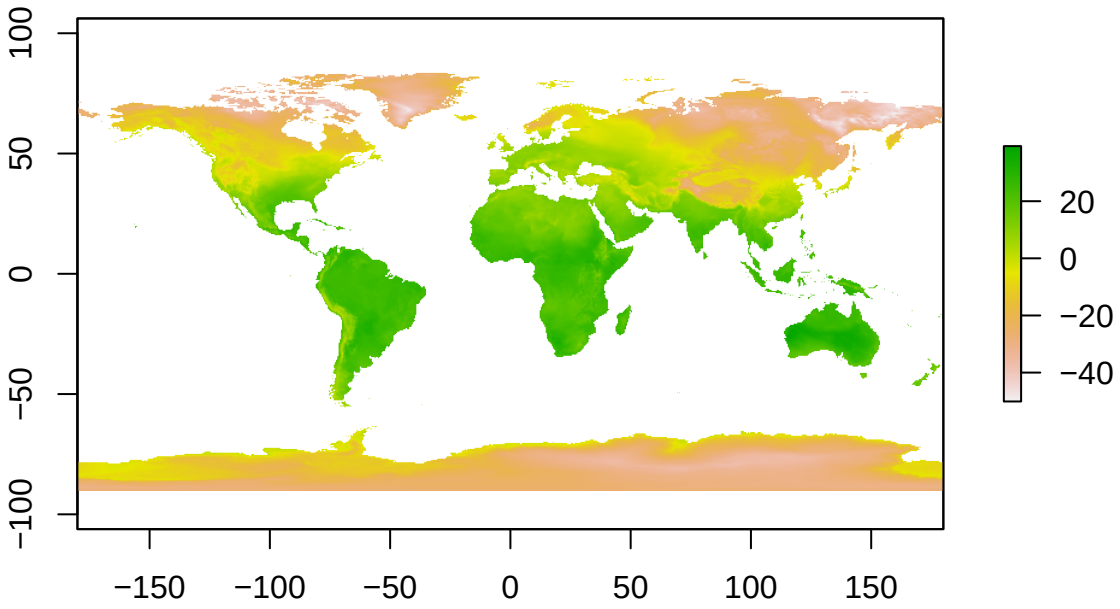
We will work with these shapefiles in next section, but first let us visualize the data in data frame we extracted for 2nd Jan, 2023. Let's visualize the data of 2nd day using the data from dataframe created above. Recall earlier in chapter 3, we used the rast function to visualize the data, so here we will convert this data frame to raster format and then visualize it.

```
# renaming the column names for rasterXYZ (function to convert dataframes to raster for visulization) c
names(temp_data_df2)<-c("x","y","temperature")

# converting it to raster projection
ras_temp<-rasterFromXYZ(temp_data_df2)

print(ras_temp)
```

```
## class      : RasterLayer
## dimensions : 360, 720, 259200  (nrow, ncol, ncell)
## resolution : 0.5, 0.5  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : NA
## source     : memory
## names      : temperature
## values     : -50.10888, 39.28652  (min, max)
```

So now raster format is same as we learned in chapter 3 (except for class, the class in step2 is spatRaster here it is Raster Layer). For our analysis, classes do not matter (we treat every class as same without going in details although one can look up the difference). Resolution is also 0.5 * 0.5, the only missing thing is CRS, so we assign the CRS (WGS84) same way as in chapter 3.

```
# assigning a coordinate system
proj4string(ras_temp)<-CRS("epsg:4326")

## plotting the data and visualizing it
plot(ras_temp)
```




```
## We will use this ras_temp raster while extracting data from it using shapefile.
```

Integrating Shapefiles for Regional Climate Data Extraction and Analysis

As mentioned earlier, the response variable (the data we need to model) in data-driven models is typically collected at standard regional levels within a country. These regions are represented by shapefiles, which contain detailed information about each region, such as the code, area, latitude-longitude coordinates, and geometry. These shapefiles are publicly available and free to use. In our context, we have already downloaded the EU NUTS3 level shapefile and the Brazil shapefile at the mesoregion level (a mesoregion is a collection of several municipalities in Brazil), and we will be using these shapefiles in our analysis. Shapefiles are already shared with everyone so we need to set the path to the directory containing these files.

Step 1: Preparing the shapefile

In this section we will demonstrates how to read, understand, assign crs, crop and visualize the the EU NUTS3 region shapefile, so that it can be further used for data extraction.

```
# first reading the shapefile from the same directory (set the directory to tutorials folders)
shp_file <- "NUTS_RG_01M_2021_3035_LEVL_3.shp"    ## name of the shapefile for EU NUTS3 region
shapefile1 <- st_read(shp_file)
```

```
## Reading layer `NUTS_RG_01M_2021_3035_LEVL_3' from data source
##   `/Users/pratik/Downloads/NUTS_RG_01M_2021_3035_LEVL_3.shp'
##   using driver `ESRI Shapefile'
## Simple feature collection with 1514 features and 9 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -2824231 ymin: -3076163 xmax: 10026010 ymax: 6405005
## Projected CRS: ETRS89-extended / LAEA Europe
```

```
#print(shapefile1)             # Detailed explanation (uncomment to print)

## now since the CRS is in Easting - Northing system
## so we convert it to standard WGS84 ( for uniform crs throughout this tutorial)
shapefile <- st_transform(shapefile1, crs = "epsg:4326")

plot(shapefile$geometry)
```

```
#Since shapefile also contain extended EU regions
# Hence, define the bounding box, so that shapefile is cropped only to main Europe
bbox <- c(xmin = -10, ymin = 29, xmax = 35, ymax = 72)

# Create the geographic extent object
cropped_region <- st_crop(x = shapefile$geometry, y = bbox)
plot(cropped_region)
```

Step 2: Aligning crs of shapefile and raster object

```
## Recall from last chapter,
##Our ras_temp raster contains the raster information for mean temperature of 2nd Jan, 2023
print(ras_temp)    ## refreshing the information
```

```
## class      : RasterLayer
## dimensions : 360, 720, 259200  (nrow, ncol, ncell)
## resolution : 0.5, 0.5  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : temperature
## values     : -50.10888, 39.28652  (min, max)
```

```
#Cropping the ras_temp to extent so that it matches with area covered by shapefile
## i.e. both have same lat long extent (as we have done in chapter 3)
temp_EU <- crop(ras_temp, extent(shapefile))
print(temp_EU)    ## now ras_temp is reduced to extent of EU NUTS3 shapefile
```

```
## class      : RasterLayer
## dimensions : 205, 238, 48790  (nrow, ncol, ncell)
## resolution : 0.5, 0.5  (x, y)
```

```
## extent     : -63, 56, -21.5, 81  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : temperature
## values     : -48.75517, 32.40563  (min, max)
```

Step 3: Extracting climate data using shapefile

We will now extract the data for each of these EU_NUTS 3 region using the shape file, for this we will use a package named exact_extract. Let's first use it and then we will look into this package in detail:

```
temp_data_NUTS3 = exact_extract(temp_EU,shapefile,
                                fun="weighted_mean",weights="area")   ## weighted by area of grid
```

```
##   |                                                                              |
```

```
#print(temp_data_NUTS3)      ## uncomment
```

Total of 1514 data points for each region and in the same order as nuts3 regions in shapefile. NAN values indicates that data for these regions in NUTS3 are not prsent in temp_EU rast object or either major portion is of region lies withing the ocean (as we are using ERA5land data which is NAN for sea coordinates).

```
temp_data_NUTS3 = as.numeric(temp_data_NUTS3)
## convert the data to numeric values (only to be double sure)
```

Step 4: Visualizing the extracted data

Now let's visualize the extracted data for each NUTS3 region with plot of these regions. Since the data is in same order as nuts3 regions, we just add a column to our shapefile containg these values

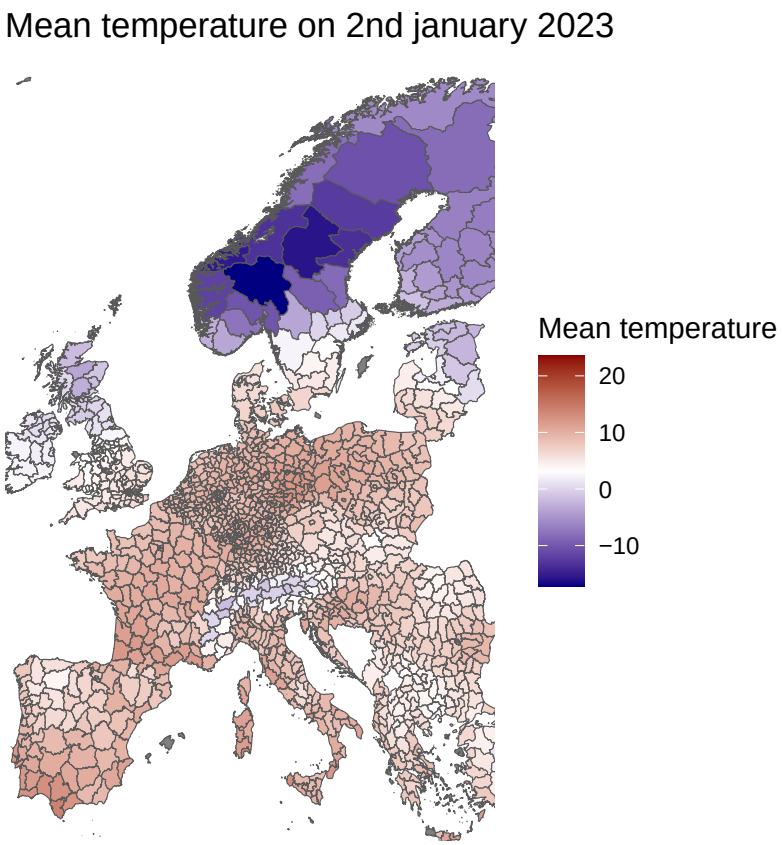
```
shapefile$values <- temp_data_NUTS3
#print(shapefile)      ## uncomment if needed to inspect
```

The values column in the shapefile contains the extracted climate data for each region, while the other columns correspond to additional information specific to the shapefile, such as region codes, names, and geometry details.

Before we move on to visualizing the extracted data, it's important to introduce the ggplot2 library, a powerful and widely-used tool for data visualization in R. Although many of us are already familiar with ggplot2, we will focus on understanding its basic structure rather than delving into its complexities. This package operates using the grammar of graphics, where plots are built by layering components such as data, aesthetics, and geometries. ggplot2 is flexible and can produce a wide range of visualizations, making it crucial for analyzing spatial and non-spatial data alike. In this section, we will explore how to use it effectively for visualizing climate data extracted at regional levels.

Let's start by plotting the extracted data, and afterward, we will break down the different arguments used in the ggplot function to understand how it works for visualizing climate data. This approach will allow us to see the visualization in action before diving into the details of how each component contributes to building the plot.

```
# Define your custom color palette with three colors
custom_colors <- c("navyblue", "white", "red4")

# Set your upper limit for legend (any value above mx temperature)
upper_limit <- 35


# Set legend title
legend_title <- "Mean temperature"

# for no space after highest value in legend

# Plot the shapefile with output_data using a custom color scale
ggplot() +
  geom_sf(data = shapefile, aes(fill = ifelse(values > upper_limit, upper_limit, values))) +
  labs(title = "Mean temperature on 2nd january 2023 ") +
  scale_fill_gradientn(colors = custom_colors, name = legend_title) +
  theme_minimal() +
  theme(
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    axis.text = element_blank(),
    axis.title = element_blank(),
    axis.ticks = element_blank()
  )+
  coord_sf(xlim = c(-10, 29), ylim = c(35, 72), expand = FALSE)
```




```
## limits for europe for main region plot only
##(other wise a plot like plot(shapefile$geometry) will appear containing extended EU region )
```

ggplot() + initializes a new ggplot plot. The empty parentheses indicate that the default plot layers will be added afterward.

geom_sf() adds a layer that plots spatial features from the shapefile object for plotting.

aes() function is used to define the aesthetic mapping, in this case, controlling the fill color of each geometry based on values

ifelse(values > upper_limit, upper_limit, values) statement limits the values being plotted. If values exceed the defined upper_limit, they are capped at upper_limit, preventing the color scale from stretching too far.

scale_fill_gradientn() creates a continuous color gradient, with custom_colors specifying the gradient colors. The legend for the fill is labeled with legend_title

theme_minimal() applies a minimalistic theme to the plot. This theme reduces visual clutter by removing non-essential background elements, creating a clean appearance

We can further customize the plot's theme further by removing the major and minor grid lines **(panel.grid.major, panel.grid.minor)**, as well as the axis text, titles, and ticks **(axis.text, axis.title, axis.ticks)**. This is often done to avoid distractions when plotting spatial data, where axes are not always relevant.

Step 5: Converting the data to working dataframes

Since visualization is completed, so now we will convert the shapefile data to a data frame for NUTS3 and save it (which can be further utilized in data driven models)

```
# extracting shapefile details in a data frame
shp_to_df <- as.data.frame(shapefile)
head(shp_to_df)       ## select only required columns
```

```
##   NUTS_ID LEVL_CODE CNTR_CODE                   NAME_LATN
## 1   NO0B2         3        NO                    Svalbard
## 2   NO0B1         3        NO                   Jan Mayen
## 3   HR064         3        HR Krapinsko-zagorska županija
## 4   DE21A         3        DE                      Erding
## 5   DE94E         3        DE        Osnabrück, Landkreis
## 6   DE94F         3        DE                      Vechta
##                     NUTS_NAME MOUNT_TYPE URBN_TYPE COAST_TYPE   FID
## 1                    Svalbard          3         3          1 NO0B2
## 2                   Jan Mayen          3         3          1 NO0B1
## 3 Krapinsko-zagorska županija          4         3          3 HR064
## 4                      Erding          4         3          3 DE21A
## 5        Osnabrück, Landkreis          4         2          3 DE94E
## 6                      Vechta          4         2          3 DE94F
##                       geometry    values
## 1 MULTIPOLYGON (((33.09131 80... -5.102651
## 2 MULTIPOLYGON (((-7.962424 7...       NaN
## 3 MULTIPOLYGON (((16.25128 46...  9.813537
## 4 MULTIPOLYGON (((12.01712 48...  6.956783
## 5 MULTIPOLYGON (((8.018153 52...  9.736039
## 6 MULTIPOLYGON (((8.459278 52...  9.802217
```

```
req_df <- shp_to_df[, c(1:9)]    # only required information (columns) from shapefile (not geometry)

df_temp_shapefile <- cbind(req_df, shapefile$values) ## bind column wise corresponding values of mean t
#print(df_temp_shapefile) # Uncomment if needed to inspect the final data frame

#Write output data to CSV file and save in your folder
write.csv(df_temp_shapefile, file = "EU_NUTS3_mean_temp.csv", row.names = FALSE) # set the directory an
```

Step 6: Extracting and visualizing the data for Brazil mesoregions using shapefile (try as an exercise without looking at solution)

Problem - Suppose we want to extract the data for meso regions of Brazil using the standard shapefile, try to do extract the data using weighted mean, plot it using ggplot and extract the data into a csv file.

End of tutorial