Technical Guideline Series

Contents

Part 5 - File Formats

Prepared by Joy Kumagai - Technical Support Unit (TSU) of Knowledge and Data

Reviewed by Aidin Niamir - Head of the Technical Support Unit of Knowledge and Data

For any inquires please contact tsu.data@ipbes.net

Version: 1.1
Last Updated: 15 July 2022

DOI: 10.5281/zenodo.6839037

I. Executive Summary

This technical guideline is for all IPBES experts and focuses on recommended file formats for IPBES with open or mostly open standards. To summarize briefly:

* While we are aiming to use open or mostly open formats, currently DOCX is widely used by the IPBES community and thus is still acceptable, although for preservation a PDF version of the DOCX file should also be added to the repository.

More details on these formats follow below. If you have any suggestions on further content or file format types you would like to see covered please contact us.

Table 1: Table 1: Recommendations for file formats

Data Type	Recommended Format
Geospatial Vector Data	GeoPackage, Shapefile, GeoJSON, KML/KMZ
Geospatial Raster Data	GeoTIFF, GeoPackage, NetCDF
Tabular Data	CSV, TXT
Textual Data	TXT, PDF, DOCX*
Figures / Line Drawings	SVG, EPS, PDF
Photographs	PNG, JPEG

II. Geospatial Files

Spatial data generally fall into either vector or raster data. Vector data consists of points, lines, and polygons that are based on point locations. Raster data is grid-based data, such as pixels on a screen or an image. Raster data is either continuous or discrete, for example a temperature dataset is continuous or a land cover dataset with classes is discrete. For more information on these geospatial data types and a general introduction to geospatial concepts please visit this data carpentry website. Additionally, for a complete list of geospatial formats please reference this guide.

For each type of spatial data, this guideline will show examples of how to export and read the information. Please download the following data using this code to follow the examples.

```
library(magrittr) # for the pipe operator
library(sf) # we use sf package to handle vector data

States <- raster::getData("GADM", country = "United States", level = 1) # Downloads data from GADM
Utah_sf <- st_as_sf(States) %>% dplyr::filter(NAME_1 == "Utah") # Select just the state of Utah for sim
```

A. Vector Data

i. GeoPackage We recommend using the GeoPackage format for storing geospatial information. This file format is new and less widely used, but a completely open format for storing geospatial data. As stated on their website "GeoPackage is an open, standards-based, platform-independent, portable, self-describing, compact format for transferring geospatial information." A GeoPackage can store both vector and raster data (as tiles) and can have multiple layers per single file. The format allows for multiple geometry types per file, so one can store both point and polygon data within the same file.

A GeoPackage is ideal for encoding geospatial data when size and power are limited such as within a mobile device and is implemented in an SQLite database. It is slightly lighter in size than a shapefile, usually around 1.1-1.3x smaller and there is not limit on file size.

We generally encourage everybody to use the GeoPackage format because of these reasons. Existing shapefiles can be converted by using the package `stars` in R and online here for small shapefiles

Some drawbacks are that the format is relatively young and the raster support in R is limited. It is very difficult to write multiband raster files using the common packages in R.

If you are interested in understanding all of the file types and contents in a GeoPackage detailed information on the structure of a GeoPackage can be found here.

To export a geopackage the following code and the `sf` package can be used:

```
Utah_sf %>%
  st_write("Utah_geopackage.gpkg", # Uses the GPKG driver of the GDAL library
           layer = "Utah")  # Names the layer, as a GeoPackage can have multiple layers
```

Next, you can read a geopackage using the simple `st_read()` function

```
Utah_test <- st_read("Utah_geopackage.gpkg",
                     layer = "Utah")
```

ii. Shapefile A shapefile is the most widely used vector type spatial data format and is recommended although it is not ideal and needs be used correctly. The geometry of a feature is stored as a shape comprising of a set of coordinates. Each feature is associated with a list of attributes within a table. Shapefiles can be

used with almost all geospatial software and are well supported by open source software libraries. It was developed and currently regulated by ESRI, a commercial company. More information on shapefiles can be found on ESRI's website here.

A shapefile consists of multiple files that together are read by a computer program which specifies geometry of features, projection, and metadata of the dataset.

All files within one shapefile share the same file name with different extensions. These files can not be separated from each other and should be zipped into a single archive when being transferred. The mandatory and some optional extensions are included below but more are described on the ESRI website here.

- .shp - Mandatory. Contains the geometry for each feature - each record describes a shape with a list of its vertices
- .shx - Mandatory. Stores the index of the feature geometry
- .dbf - Mandatory. Dataset that stores the attribute information of feature with a one-to-one relationship between geometry and attribute rows
- .prj - Necessary. Stores the metadata associated with the shapefiles coordinate and projection system. This file needs to be included or the data can not be used correctly
- .xml - Optional. Contains the metadata associated with the shapefile
- .sbn and .sbx - Optional. Two spatial index files that optimize spatial queries. These two files make up a shape index to speed up spatial queries
- .cpg - Optional. Describes the encoding applied to create the shapefile

Some drawbacks to using shapefiles are the following:

- Not a completely open format
- Lacks support for UNICODE character strings, therefore limiting the use of non-English languages
- Consists of multiple files which can easily be separated
- Field names can only be 10 characters or shorter in length
- Size limit of 2GB
- Can only have one type of geometry per file (only point data or only polygon data)
- Does not store geometry of features. e.g. polygons which are next to each other are independent and joining borders are coded as separate line segments, which can result in holes and islands.

To export and read a shapefile, the **sf** package can also be used.

```
# Export
Utah_sf %>%
  write_sf("Utah_shapefile.shp")

# Read
Utah_test2 <- read_sf("Utah_shapefile.shp")
```

iii. GeoJSON Another vector format if the recommended formats are not possible is GeoJSON. GeoJSON is a simple open standard geospatial format that also represents features and associated attributes. This format is commonly used in web-based mapping. It is based on JavaScript Object Notation (JSON) and the standard format uses a geographic coordinate reference system, WGS 1984. Unlike shapefiles, it was not developed by a commercial company, but an internet working group of developers and thus is openly documented.

We recommend only using this format when data is simple points and lines as it is text based. Since it is text based, it is easy for humans to read directly and for machines to parse through and almost all GIS programs used for applications on the web can write and read GeoJSON data.

An important drawback worth mentioning is that it lacks support for projections. Additionally, there is no built-in support for embedding rich metadata about the dataset as a whole. Therefore, when using GeoJSON, provide the additional structured metadata with the json file so that the data is interoperable including an additional metadata file which specifies the projection and map datum.

More technical information can be found here on the Sustainability of Digital Formats website which supports the US Library of Congress Collections.

To export and read a GeoJSON file the following code can be used. The option `RFC7946 = YES` needs to be used when exporting as it is a more recent and strict standard for GeoJSON.

```
# Export
Utah_sf %>%
  st_write("Utah.geojson",
           driver = "GeoJSON",
           layer_options = "RFC7946=YES")

# Read
Utah_test3 <- read_sf("Utah.geojson")
```

iv. KML/KMZ KML (Keyhole Markup Language) is a geospatial publishing format that enables easy visualization. The format is used primarily in the Google Earth Interface. The KML format focuses on visualization and allows one to encode what information to show and how to show it including annotation of images and maps and supports 3D textured models. KML supports display of rich data through icons and captions and can control the users view point directing where to look.

KML files can be created and edited on the google earth interface or can be drafted in an XML or simple text editor. When KML files are shared usually they are compressed and zipped into KMZ files. There are various ways to package a KML file, and thus we recommend other formats for storing or transferring geospatial data.

KML only uses one coordinate reference system and is not well suited for delivering large quantities of data.

More information on KML is available here from Google.

To export a KML file from R, the following code can be used. First, the projection needs to be checked as KML only supports WGS84.

```
st_crs(Utah_sf) # check projection
```

```
## Coordinate Reference System:
##   User input: +proj=longlat +datum=WGS84 +no_defs
##   wkt:
## GEOGCRS["unknown",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
```

```
##             LENGTHUNIT["metre",1]],
##         ID["EPSG",6326]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433],
##         ID["EPSG",8901]],
##     CS[ellipsoidal,2],
##         AXIS["longitude",east,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433,
##                 ID["EPSG",9122]]],
##         AXIS["latitude",north,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433,
##                 ID["EPSG",9122]]]]
```

Next, the same functions `st_write()` and `st_read()` can be used

```
# Export
st_write(Utah_sf, "Utah.kml", driver = "kml")

# Read
Utah_test4 <- st_read("Utah.kml")
```

B. Raster Data

i. GeoTIFF and Cloud Optimized GeoTIFF Raster data can come in a variety of formats such as png, tiff, or jpeg, commonly used for images, but here we focus on geospatial data formats. We recommend GeoTIFF for geospatial raster data. GeoTIFF is a formatted TIFF 6.0 raster file that embeds cartographic information into the raster image as tags. The format was originally developed as a format to distribute satellite or aerial photography imagery and is widely used.

There are many benefits to using the GeoTIFF format. There is strong software support in the form of open source libraries and many commercial and open GIS and spatial data analysis software products support reading and writing GeoTIFF data. It is highly interoperable, used worldwide, can store multiband raster data, and supported for many years.

The tags of GeoTIFF files, called `tif tags` should include the following metadata: extent, resolution, datum, projection (CRS), and values that represent missing data. These tags are incredibly important and are how programs recognize the spatial coverage and projection of the raster data.

GeoTIFF is not suitable for storing complex multi-dimensional data structures and has a size limit of 4GB.

More information can be found on here from the NASA standards and references webpage.

To export a GeoTIFF file, the following code can be used with the `raster` package.

```
library(raster)

# Create raster to export
artwork <- raster() %>%
  setValues(runif(ncell(.)))  # fill with random values

# Export
writeRaster(artwork, "artwork.tif")
```

To read a GeoTIFF raster, just one line of code is needed

```
raster("artwork.tif")
```

```
## class      : RasterLayer
## dimensions : 180, 360, 64800  (nrow, ncol, ncell)
## resolution : 1, 1  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : artwork.tif
## names      : artwork
## values     : 2.358807e-06, 0.9999586  (min, max)
```

Recently, a format called, Cloud Optimized GeoTIFF, has become increasingly popular which is a regular GeoTIFF file aimed at being hosted on a HTTP file server. This standard was developed in 2016 within the Open Source Geospatial Foundation project. The format enables efficient workflows on the cloud by utilizing tiling and overviews. It allows for efficient imagery data display and access through HTTP GET range requests, so end-users can just use the parts of the GeoTIFF they need. A cloud optimized GeoTIFF is larger in size than a normal GeoTIFF, but it enables faster access on a server.

If you are interested in exporting Cloud Optimized GeoTIFFs, one option is to use the `write_tif` function from the `gdalcubes` package described here

ii. Geopackage (raster) We recommend using the GeoPackage format for storing raster data as well. Raster data is stored in a tile-based pyramid structure within the GeoPackage, therefore the imagery or raster information is stored at multiple resolutions. The parameters of the tiles can be set when writing the layer to the Geopackage. This tile-based pyramid structure is useful when handling a GeoPackage on a small device, as the appropriate resolution can be displayed based on the zoom level and screen size.

More information on how the tiles are stored within the GeoPackage can be found here

Please note that in R, currently support for writing multiband rasters in a GeoPackage is limited.

To write and read a GeoPackage with a raster layer within it, the following code can be used with the package `stars`

```
library(stars)

# Writing a Geopackage
artwork %>%
  st_as_stars %>% # this converts the RasterLayer to a stars object
  write_stars("artwork.gpkg",
              driver = "GPKG")

# Read the GeoPackage
artwork_gpkg_stars <- read_stars("artwork.gpkg") %>%
  as("Raster")
artwork_gpkg_stars
```

```
## class      : RasterLayer
## dimensions : 180, 360, 64800  (nrow, ncol, ncell)
## resolution : 1, 1  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer
## values     : 2.358807e-06, 0.9999586  (min, max)
```

iii. NetCDF Another option for storing geospatial raster data is NetCDF (Network Common Data Form) if a GeoTIFF will not suit your purpose or you were given data in NetCDF format. NetCDFs are often used for climate and large scientific raster data files, especially for storing multidimensional scientific data. NetCDF is used by a large community and is self-describing, portable, scalable, appenable, archivable, and is considered a standard. It is in the public domain, and thus open, well documented, and actively developed and maintained. NetCDF files support multidimensional arrays with multiple unlimited appendable dimensions but is not as commonly used as GeoTIFFs.

Some limitations to NetCDFs are that they are not as user-friendly to work with especially in R, they do not support nested structures, and no real compression is supported.

NetCDF was developed and maintained at Unidata. On their website here they provide tutorials and documentation. A quick factsheet with more information is available here.

R is able to read and write netCDF files using the package `ncdf4`. For a full tutorial on how to read and write netCDF files, please refer to this website

We will now go through a very simple example starting with formatting and exporting NetCDF data adapted from this guide and finally importing the netCDF we create. We will use the base R volcano dataset.

```
library(ncdf4)
data(volcano) # store volcano dataset
```

Next, we will store the elevations and grid dimensions in variables.

```
z <- 10*volcano       # matrix of elevations
x <- 100*(1:nrow(z)) # meter spacing (S to N)
y <- 100*(1:ncol(z)) # meter spacing (E to W)
```

We will now define the spatial dimensions of the data (lat / long) and then use ncvar_def to define a variable in the netCDF file that will hold the elevation data.

```
# Define the netcdf coorindate variables
dim1 <- ncdim_def(name = "EW", units = "meters", vals = as.double(x)) # Defines longitude
dim2 <- ncdim_def(name = "SN", units = "meters", vals = as.double(y)) # Defines latitude

# Define the empty netcdf variable (elevation)
fillvalue <- 1e32
dlname <- "Elevation"
elevation_def <- ncvar_def("Elevation", units = "meters", list(dim1,dim2), fillvalue, dlname , prec="dou
```

We will now create the netCDF file and add the variables into the file. At this step, one can add additional metadata such as title, affiliated institution, source, references, etc.

```
# create netCDF file and put arrays
file_volcano <- nc_create("Volcano.nc",list(elevation_def),force_v4=TRUE)

# Add variables into the file
ncvar_put(file_volcano, elevation_def, z)

# Get a summary of the created file:
file_volcano
```

```
## File Volcano.nc (NC_FORMAT_NETCDF4):
```

```
##
##      1 variables (excluding dimension variables):
##         double Elevation[EW,SN]   (Contiguous storage)
##             units: meters
##             _FillValue: 1e+32
##
##      2 dimensions:
##         EW  Size:87
##             units: meters
##             long_name: EW
##         SN  Size:61
##             units: meters
##             long_name: SN
```

Finally, close the file, which writes the data to disk.

```
nc_close(file_volcano)
```

Now, let's move onto opening the netCDF file we just created.

```
# First step is to open the file
example <- nc_open("Volcano.nc")
example
```

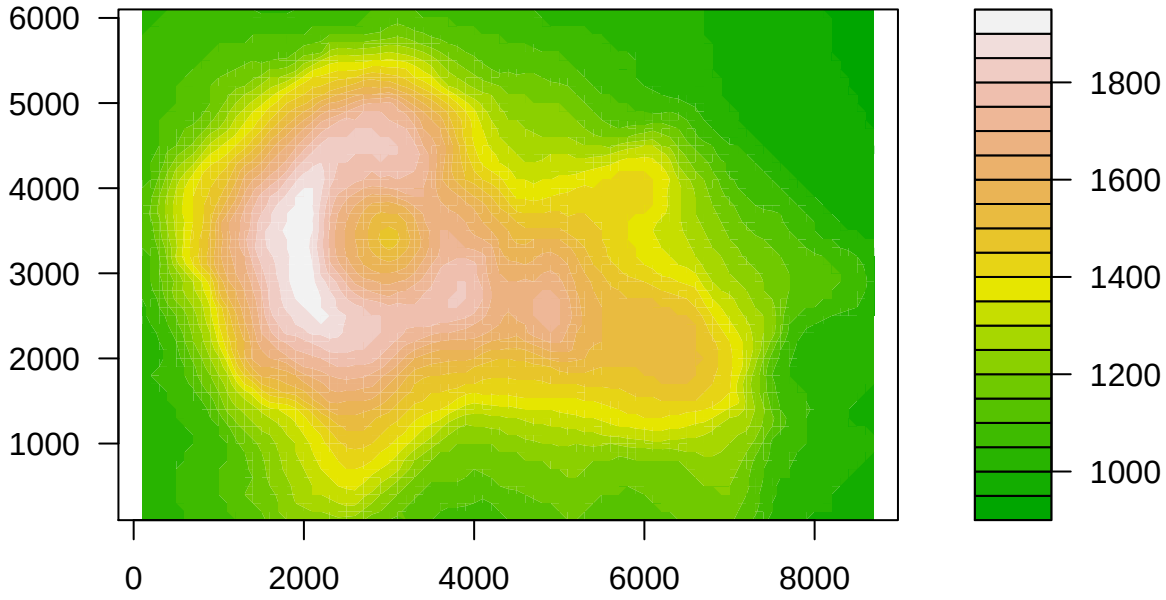
```
## File Volcano.nc (NC_FORMAT_NETCDF4):
##
##      1 variables (excluding dimension variables):
##         double Elevation[EW,SN]   (Contiguous storage)
##             units: meters
##             _FillValue: 1e+32
##
##      2 dimensions:
##         EW  Size:87
##             units: meters
##             long_name: EW
##         SN  Size:61
##             units: meters
##             long_name: SN
```

Next, we will extract the coordinate variables and elevation variable.

```
lon <- ncvar_get(example,"EW")               # longitude
lat <- ncvar_get(example, "SN")              # latitude
elevation <- ncvar_get(example, "Elevation") # variable
```

Now, one can create a plot from the netcdf file.

```
filled.contour(lon,lat,elevation, color = terrain.colors, asp = 1)
```



C. Resources and useful R packages for handling spatial data

For more information on these geospatial data types and other types can be found here as well: https://gisgeography.com/gis-formats/

While information on the various formats for spatial data are incredibly important to understand when using and sharing geospatial data, we would also like to recommend a few packages for handling spatial data in R.

The `sf` package is used to handle vector data, while the `terra` and `raster` packages are often used to handle raster data. The packages `sf` and `raster` have been used throughout these technical guidelines. For an overview on these packages, I recommend the vignettes on R Spatial for each of the packages linked below.

- sf
- raster
- terra

Finally, here is a table summarizing the recommended r packages for each data type with links to more guides on how to read, convert, and write in R between these formats:

* Important note: The commonly used package `raster` will be replaced by the new package `terra` as `rgdal`, one of the key packages it uses, will be retired in 2024. There are many more resources available online to guide people on how to use these packages than those mentioned.

Table 2: Table 2: Recommendations for handling these file formats in R

Data Type	Recommended R Package	Link to a converstion guide
GeoPackage (vector)	sf / stars	https://inbo.github.io/tutorials/tutorials/spatial_standards_vector/
GeoPackage (raster)	sf / stars	https://inbo.github.io/tutorials/tutorials/spatial_standards_raster/
Shapefiles	sf	https://datacarpentry.org/r-raster-vector-geospatial/06-vector-open-sh
GeoJSON	sf	https://inbo.github.io/tutorials/tutorials/spatial_standards_vector/
KML/KMZ	sf	https://stackoverflow.com/questions/57415464/label-kml-features-usin
GeoTIFF	raster / terra	https://inbo.github.io/tutorials/tutorials/spatial_standards_raster/
NetCDF	ncdf4	https://pjbartlein.github.io/REarthSysSci/netCDF.html

Table 3: Table 3: Example of wide data

Experiment	Minimum Temperature	Maximum Temperature	Average Temperature
1	8	20	15
2	15	27	23
3	21	30	25

III. Tabular Files

The rest of this technical guideline will go through recommended file formats for tabular, textual files, and image files with less detailed information.

i. Structure Often when one pictures data, they don't think of complex geometries, but rather tabular data such as a list of field sites and associated variables recorded in an table or a survey responses in an excel sheet. Tabular data is data structured into rows and columns and can be wide or long.

Wide data is where each different variable is listed in a separate column, while long data is structured so a row is one observation, therefore one column contains all values, and another column contains the variable. Often with small amounts of data, a wide table can be more easily interpreted by humans, while long data is often required for running statistical analysis in computer programs. The use of the long format is highly recommended!

We recommend the following open file formats for tabular data which does not include Excel sheets.

ii. CSV / TXT A CSV file (Comma Separated Variable) file is a text based data file that describes tabular data. Each row is one line, and the columns are separated by commas.

Table 4: Table 4: Example of long data

Experiment	Variable	Temperature
1	Minimum_Temperature	8
1	Maximum_Temperature	20
1	Average_Temperature	15
2	Minimum_Temperature	15
2	Maximum_Temperature	27
2	Average_Temperature	23
3	Minimum_Temperature	21
3	Maximum_Temperature	30
3	Average_Temperature	25

It is widely used as a data exchange format and highly recommended. If your tabular data do not contain commas, than a CSV file is the recommended choice. If your data do contain commas, such as survey responses, one can delineate columns in a TXT file based on tab's or other characters such as semicolons.

A CSV file can be created from an excel sheet easily described here, written in a txt file, or created in R or other computer programming languages. Text files are highly interoperable and can be imported and exported by almost any software designed for storing or manipulating data.

To export a table as a CSV file or tab delineated file in R, follow and adapt this code:

```
# csv
write.csv(mtcars,                    # Dataset
          "mtcars.csv",              # Name of file
          fileEncoding="UTF-8")      # Specifies encoding (UTF-8 preferred)

# tab
write.table(mtcars,                  # Dataset
            file = "mtcars.txt",     # Name of file
            sep = "\t",              # Tab delineation
            fileEncoding = "UTF-8") # Specifies encoding (UTF-8 preferred)
```

IV. Textual Files

For files mostly comprising of text, such as transcripts and survey questions, we recommend .txt, .pdf, or .docx files which are explained in more detail below. While .docx are not open and therefore not ideal, currently the IPBES communitiy relies on them heavily and thus this format is still recommended. When using .docx, also export the document to a pdf when submitting the data to a repository.

i. TXT A .txt file, is a plain text file that just contains text, therefore it is human-readable and interoperable. This file can be opened in any text editor and almost all operating systems come with text editors that allow you to easily create, open and edit text files. There is no limitation on the size of contents. The default character set of text files is ASCII, but they can also be saved in Unicode format, such as UTF-8 which is acceptable.

ii. PDF A PDF (Portable Document Format) was originally created by Adobe back in the 1990s, but was released as a open format in 2008. PDFs allow one to view documents easily independent of application software and operating system that is why presentations are often shared in PDF format as opposed to a powerpoint file. The PDF file format has the ability to contain not only text, but images, hyperlinks, form-fields, digital signatures, attachments, and other information. PDFs are very suitable for long-term storage of documents, as they are independent of application software.

More information on the PDF file format can be found here.

iii. DOCX DOCX files are Microsoft Word documents. This format is not open and is not interoperable. While TXT and PDF files are preferred to DOCX files, the IPBES community relies heavily on DOCX files and thus it is still acceptable to use this format. In 2007, the original standard DOC file format was updated to a new standard DOCX file, where the addition of the X stands for XML. When using .docx, also export the document to a pdf when submitting the data to a repository.

If you interested, more technical information can be found here.

V. Image Files

There are many different image file formats that one can choose from when exporting images. We recommend commonly used formats that are open and interoperable, such as SVG or PNG, but other formats may also be appropriate depending on the use case. Similar to geospatial information, there are two image file types vector and raster. For a more comprehensive list of image file formats please see this website

Vector formats are superior to raster formats for most images such as line drawings, including plots, graphs, or logos. Vector formats will never look pixelated, as they are not based on pixels, but rather mathematical forumlas that define geometric objects such as polygons, lines, and curves. Vector images are more malleable than raster images, smaller in size, faster to display, and perfectly scalable.

Raster images are based on pixels with a defined resolution and are the preferred format for photographs or non-line art images. One of the largest considerations for raster image files is resolution. Resolution is often reported in the units DPI, dots per inch. As resolution increases, clarity and the size of the file increases. The standard for displaying images on websites is 72dpi, for printing it is 300dpi or higher, and for submitting figures for publications it is 600dpi.

Vector Image Files

i. SVG SVG (Scalable Vector Graphics) is a vector image format, which uses XML text to specify lines and color. SVG is a great option and highly recommended for graphs, logos, and illustrations, especially for publishing materials on the internet. It is supported by all major browsers, but most default image editors do not support SVG. It should not be used to save photographs.

To export a SVG file from R, one can use this base code and add additional arguments such as height, width, point size, to the `svg()` function.

```
svg("example_1.svg")                      # SVG graphics device and file name

plot(rnorm(100), main = "Example Graph") # Plot your graph

dev.off()                                 # Close the graphics device
```

ii. EPS EPS (Encapsulated PostScript) was created by Adobe in 1992 and is based on Postscript rather than XML. EPS was originally intended for a print workflow not an online workflows and is no longer in development. EPS file format is recommended and better than SVG for high-quality document printing, printed logos, and marketing materials.

To export a EPS file from R, one can use this base code and add additional arguments to the `postscript()` function.

```
setEPS()
postscript("example_2.eps")

plot(rnorm(100), main = "Example Graph")  # Plot your graph

dev.off()
```

iii. PDF PDFs (Portable Document Files) mentioned previously under the textual files section can also be saved as a vector file. Vector formated PDFs allow one to easily select objects and are preferred to raster formatted PDFs. PDFs can have a mix of vector and raster content, but when exported from R, the graphics will be in vector format. If scanned, the PDF will be in raster format.

To export a figure as a PDF file from R, one can use this base code and change and add additional arguments to the `pdf()` function.

```
pdf("example_3.pdf",
    width = 4, height = 4,              # Width and height in inches
    paper = "A4")                       # Paper size

plot(rnorm(100), main = "Example Graph") # Plot your graph

dev.off()
```

Raster Image Files

i. PNG PNG (Portable Network Graphics) files are a type of raster file format that supports lossless data compression and has no copyright limitations. We generally recommend PNG files for storage and sharing images online. The lossless compression means that there is no loss in quality each time it is opened and saved again. PNG only supports the RGB color space and not CMYK, and does not support animations.

To export a PNG file from R, one can use this based code and change and add additional arguments to the `png()` funciton.

```
png("example_4.png",
     width = 4, height = 4,
     units = "in",                      # Units in inches
     res = 300)                         # resolution in dpi

plot(rnorm(100), main = "Example Graph") # Plot your graph

dev.off()
```

ii. JPEG / JPG In comparison, JPEG (or JPG) files are raster files that are often used online for displaying images as they are fast to load. It has lossy compression which means each time it is saved it reduces in file size but also in quality. We do not recommend JPEG files for long term storage, except in the case of images which come from e.g. cameras as nothing is gained in converting an original JPEG into a PNG.

To export a jpeg file from R, one can use this base code and change and add additional arguments to the `jpeg()` argument, although we discourage exporting graphs in jpeg.

```
jpeg("example_5.jpeg", width = 4, height = 4, units = 'in', res = 300)
plot(rnorm(100), main = "Example Graph")                              # Plot your graph
dev.off()
```

iii. TIFF Another option is TIFF. A TIFF (Tagged Image File) is a raster file that also supports lossless compression. We do not recommend using this tile type on websites as it is slow to load due to its large size and has limited browser support, but it is recommended for long term storage or publications. If there are no concerns about losing embeded metadata and tags, in general, it makes sense to convert a TIFF image to PNG as it reduces the size and is lossless.

To export a TIFF image file from R, one can use this base code and change and add additional arguments to the `tiff()` function, although we discourage exporting graphs in tiff.

```
tiff("example_6.tiff", width = 4, height = 4, units = "in", res = 300)
plot(rnorm(100), main = "Example Graph")                                    # Plot your graph
dev.off()
```

Thank you for your time. If you have any suggestions on further content or file format types you would like to see covered please contact us at the technical support unit at tsu.data@ipbes.net.