Introduction to the rasterbc package

Dean Koch

2021-11-18

Installation

This package is still in development, but a release on CRAN is planned in the near future. For now the package may be tested by installing the `devtools` package (run `install.packages('devtools')`), and then running the following two lines:

```
library(devtools)
install_github('deankoch/rasterbc')
```

I have tried to keep dependencies to a minimum. The package requires `sf` and `raster` for loading and merging geospatial data. If these are not already installed on your machine, the `install_github` line will ask to install them. We also use the `bcmaps` package in this vignette to define a study region (but it's not a requirement of the package).

```
library(sf)
library(raster)
library(bcmaps)
library(rasterbc)
```

Local data storage

`rasterbc` is a data-retrieval tool. Start by setting a storage directory for the raster layers

```
# replace 'H:/rasterbc_data' with your own path
datadir_bc('H:/rasterbc_data', quiet=TRUE)
```

When `quiet=FALSE` (the default), the function will ask users to confirm that `rasterbc` should write files to the supplied directory, and warn if this directory contains any existing files/folders. Note that if the storage directory has existing files with names matching those fetched by the `rasterbc` package, those data can be overwritten. Note that files are only overwritten via calls of the form `raster::getdata_bc(..., force.dl=TRUE)` (the default is `force.dl=FALSE`).

However, to be safe you should set the data directory to a path that won't used by other applications; *eg.* a subfolder of your home directory or external storage device, with a unique folder name. In future sessions, users can set `quiet=TRUE` to skip the interactive prompt and suppress warnings about existing data.

This path string is stored as an R option. View it using:

```
datadir_bc()
#> current data storage path: H:/rasterbc_data
```

Depending on the geographical extent of interest and the number different layers requested, the storage demands can be high. For example, if every layer is downloaded, then around 30 GB of space is needed. Make sure you have selected a drive with enough free space for your project.

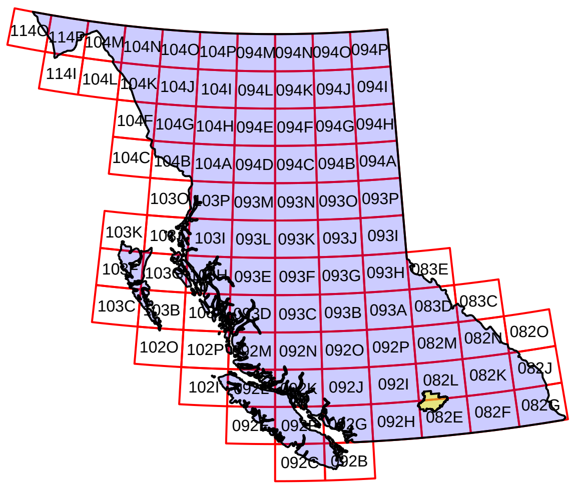
Getting started

To demonstrate this package we'll need a polygon covering a (relatively) small geographical extent in BC. Start by loading the bcmaps package and grabbing the polygons for the BC provincial boundary and the Central Okanagan Regional District

```r
# define and load the geometry
example.name = 'Regional District of Central Okanagan'
bc.bound.sf = bc_bound()
districts.sf = regional_districts()
example.sf = districts.sf[districts.sf$ADMIN_AREA_NAME==example.name, ]

# plot against map of BC
plot(st_geometry(ntspoly_bc), main=example.name, border='red')
plot(st_geometry(bc.bound.sf), add=TRUE, col=adjustcolor('blue', alpha.f=0.2))
plot(st_geometry(example.sf), add=TRUE, col=adjustcolor('yellow', alpha.f=0.5))
text(st_coordinates(st_centroid(st_geometry(ntspoly_bc))), labels=ntspoly_bc$NTS_SNRC, cex=0.5)
```

Regional District of Central Okanagan



The Okanagan polygon is shown in yellow, against a red grid that partitions the geographic extent of the province into 89 smaller regions, called *mapsheets*. This is the NTS/SNRC grid used by Natural Resources Canada for their topographic maps, with each mapsheet identied by a unique number-letter code. `rasterbc` uses this grid to package data into blocks for distribution. It is lazy-loaded as the `sf` object `ntspoly_bc`:

```r
print(ntspoly_bc)
#> Simple feature collection with 89 features and 1 field
#> Geometry type: POLYGON
#> Dimension:     XY
#> Bounding box:  xmin: 199960.5 ymin: 331658 xmax: 1874986 ymax: 1745737
#> Projected CRS: NAD83 / BC Albers
#> First 10 features:
#>    NTS_SNRC                       geometry
#> 1      092B POLYGON ((1299175 340112.5,...
#> 2      092C POLYGON ((1149647 333772, 1...
#> 3      092E POLYGON ((854708.4 444669.3...
#> 4      092F POLYGON ((1001221 442634.1,...
#> 5      092G POLYGON ((1147733 444738.3,...
#> 6      092H POLYGON ((1294127 450980.4,...
#> 7      083C POLYGON ((1548365 807589.3,...
#> 8      083D POLYGON ((1411944 794023.1,...
#> 9      083E POLYGON ((1402477 905139.5,...
#> 10     082E POLYGON ((1440286 461355.4,...
```

A basic example

Let's download Canada's 1:250,000 digital elevation model (CDEM) layers corresponding to the yellow polygon. For the full BC extent, these rasters would occupy around 1.2GB of space. But we only want the smaller extent corresponding to the polygon. There are three blocks (totalling about 20 MB) which overlap with our region of interest

```
findblocks_bc(example.sf)
#> [1] "092H" "082E" "082L"
```

fetch them using the command:

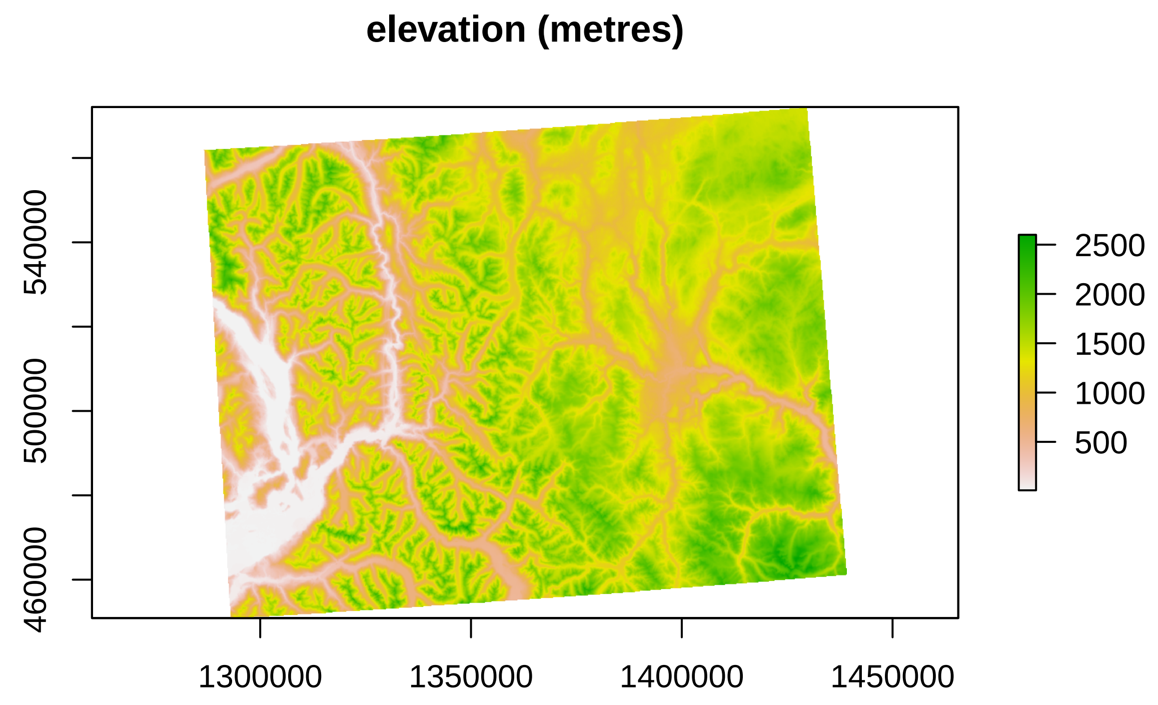
```
getdata_bc(geo=example.sf, collection='dem', varname='dem', load.mosaic=FALSE)
```

You should see progress bars for a series of three downloads, and once finished, the paths of the downloaded files are printed to the console. Note that if a block has been downloaded already (*eg.* by a `getdata_bc` call with a different `geo` argument), the existing copy will be detected, and the download skipped. *eg.* repeat the call. . .

```
getdata_bc(geo=example.sf, collection='dem', varname='dem', load.mosaic=FALSE)
#> all 3 block(s) found in local data storage. Nothing to download
#> [1] "H:/rasterbc_data/dem/blocks/dem_092H.tif"
#> [2] "H:/rasterbc_data/dem/blocks/dem_082E.tif"
#> [3] "H:/rasterbc_data/dem/blocks/dem_082L.tif"
```

. . . and nothing happens, because the data are there already. Verify by loading one of the files as `RasterLayer`:

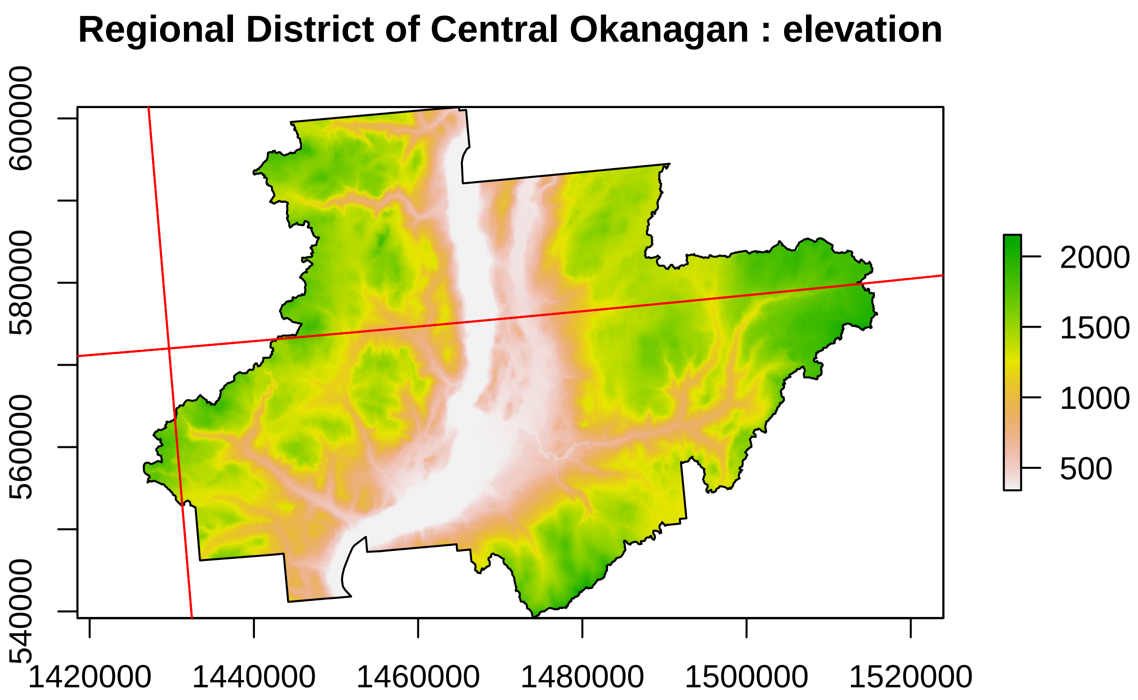
```
example.raster = raster('H:/rasterbc_data/dem/blocks/dem_092H.tif')
print(example.raster)
#> class      : RasterLayer
#> dimensions : 1212, 1525, 1848300  (nrow, ncol, ncell)
#> resolution : 100, 100  (x, y)
#> extent     : 1286588, 1439088, 450888, 572088  (xmin, xmax, ymin, ymax)
#> crs        : +proj=aea +lat_0=45 +lon_0=-126 +lat_1=50 +lat_2=58.5 +x_0=1000000 +y_0=0 +datum=NAD83
#> source     : dem_092H.tif
#> names      : dem_092H
#> values     : 7.653875, 2608.961  (min, max)
plot(example.raster, main='elevation (metres)')
```



Loading/merging blocks

To display the elevation data for the entire district, we need to combine the three blocks downloaded earlier. This can be done using `getdata_bc` with `load.mosaic=TRUE` (the default setting), which loads all required blocks, merges them into a single layer, crops and masks as needed, and then loads into memory the returned `RasterLayer` object:

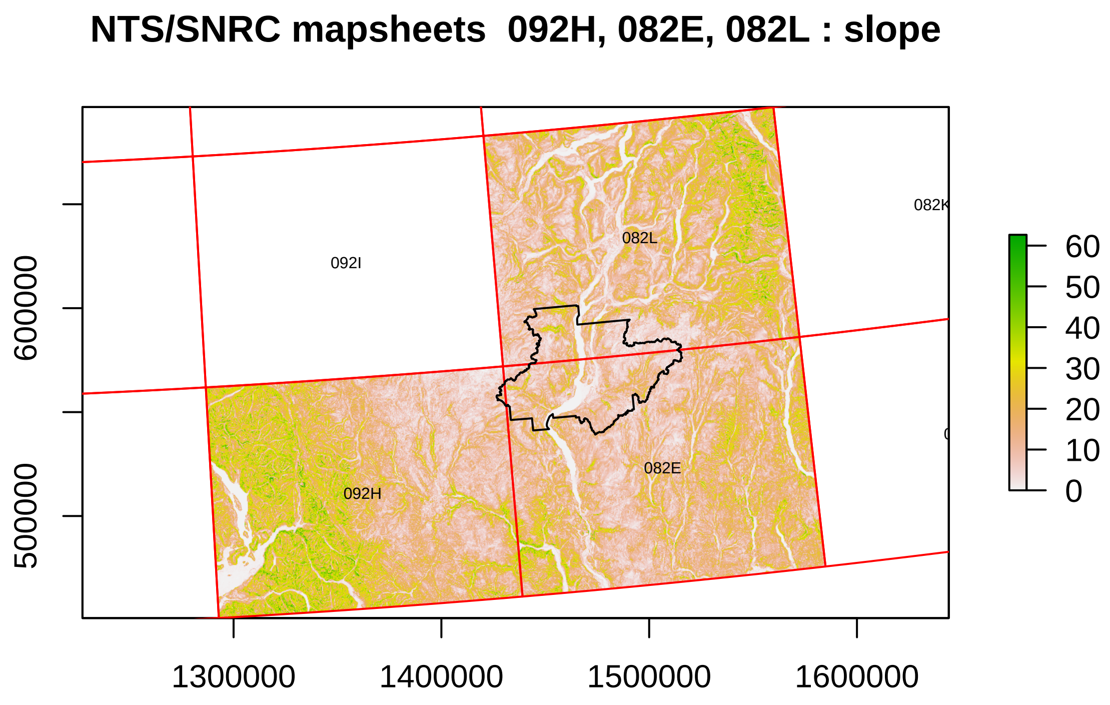
```
example.tif = getdata_bc(example.sf, collection='dem', varname='dem')
#> all 3 block(s) found in local data storage. Nothing to download
#> creating mosaic of 3 block(s)
#> clipping layer...masking layer...done
print(example.tif)
#> class      : RasterLayer
#> dimensions : 622, 893, 555446  (nrow, ncol, ncell)
#> resolution : 100, 100  (x, y)
#> extent     : 1426588, 1515888, 539188, 601388  (xmin, xmax, ymin, ymax)
#> crs        : +proj=aea +lat_0=45 +lon_0=-126 +lat_1=50 +lat_2=58.5 +x_0=1000000 +y_0=0 +datum=NAD83
#> source     : memory
#> names      : dem
#> values     : 340.43, 2153.29  (min, max)
plot(example.tif, main=paste(example.name, ': elevation'))
plot(st_geometry(example.sf), add=TRUE)
plot(st_geometry(ntspoly_bc), add=TRUE, border='red')
```



Any simple features object of class `sf` or `sfc` can be used for the argument `geo`, provided its geometry intersects with the provincial boundary of BC. The intended usage is for the user to delineate their region of interest as a `(MULTI)POLYGON` object (here, `example.sf` is a `MULTIPOLYGON`). Geometries of other classes (such `SpatialPolygons`, as defined by `sp`; or data frames containing coordinates of vertices) can often be coerced to `sf` using a command like `sf::st_as_sf(other_geometry_class_object)`.

Alternatively, users can directly download individual blocks by specifying their their NTS/SNRC codes, *eg.* here is the "slope" variable (from the "dem" collection), specified using the codes:

```
example.codes = findblocks_bc(example.sf)
example.tif = getdata_bc(example.codes, collection='dem', varname='slope')
#> all 3 block(s) found in local data storage. Nothing to download
#> creating mosaic of 3 block(s)
#> loading block(s)...done
plot(example.tif, main=paste('NTS/SNRC mapsheets ', paste(example.codes, collapse=', '), ': slope'))
plot(st_geometry(ntspoly_bc), add=TRUE, border='red')
plot(st_geometry(example.sf), add=TRUE)
text(st_coordinates(st_centroid(st_geometry(ntspoly_bc))), labels=ntspoly_bc$NTS_SNRC, cex=0.5)
```



File management

If you forget which files have been downloaded, you can either check the directory `data.dir` using your file browser (subfolder '/dem/blocks', in this case), or use `listfiles_bc` to get a logical vector indicating which files are curerntly found in your local storage directory:

```
is.downloaded = listdata_bc(collection='dem', varname='dem', simple=TRUE)
sum(is.downloaded)
#> [1] 3
length(is.downloaded)
#> [1] 89
```

This shows that of the 89 blocks for the variable name 'dem' (in the collection 'dem'), we have downloaded three so far. Notice the return value of `listdata_bc` is a *named* vector, with names indicating the destination filenames and paths. This shows where they will be written by `getdata_bc`. All filenames are either of the form 'varname_mapsheet.tif' (as in this example) or else varname_year_mapsheet.tif (for time-series data).

By default, the `listdata_bc` function prints a list of all available layers. *eg.* in the 'dem' collection we also have 'aspect' and 'slope':

```
listdata_bc(collection='dem', verbose=2)
#>        year                    description                                       unit tiles
#> dem      NA                 digital elevation map                  (metres above sea level)  3/89
#> slope    NA derived from digital elevation map               (degrees above horizontal)  3/89
#> aspect   NA derived from digital elevation map (degrees counterclockwise from north)  0/89
```

Notice the 'slope' blocks that were downloaded manually using NTS/SNRC codes. We merged these blocks earlier in the `opendata_bc` function call that created `example.tif`. Currently, this layer resides in memory and can be accessed via the R object `example.tif`. To save a copy, one can use the `raster::writeRaster` function:

```
slope.path = file.path(getOption('rasterbc.data.dir'), 'dem', 'example_slope.tif')
writeRaster(example.tif, slope.path, overwrite=TRUE)
```

`getdata_bc` writes all of its data inside a 'blocks' subdirectory (in this case '/dem/blocks'), and the subfolder of the data directory corresponding to the collection (in this case '/dem') is, by default, left empty. So it is a good place to store and organize such derivative files, where they can be loaded more quickly (in future), *eg.*

```
raster(slope.path)
#> class      : RasterLayer
#> dimensions : 2459, 2983, 7335197  (nrow, ncol, ncell)
#> resolution : 100, 100  (x, y)
#> extent     : 1286588, 1584888, 450888, 696788  (xmin, xmax, ymin, ymax)
#> crs        : +proj=aea +lat_0=45 +lon_0=-126 +lat_1=50 +lat_2=58.5 +x_0=1000000 +y_0=0 +datum=NAD83
#> source     : example_slope.tif
#> names      : example_slope
#> values     : 0, 67.20546  (min, max)
```

If you're finished with `rasterbc` and want to remove all of the stored data, or if you simply want to free up space, the entire data directory or any of its contents can be deleted using your file browser. This will not break the `rasterbc` installation. However, all downloaded data will be erased and you will need to run `datadir_bc` again before using the other package functions.

Integer codes

Note that the `bgcz` collection data are factors, which are then encoded in the geotiff files as integer codes. `opendata_bc` returns these factor names in a raster attribute table for the `RasterLayer` object (column "code"). The complete lookup tables are also stored in the lazy loaded list object `metadata_bc`.

```
lookup.list = rasterbc::metadata_bc$bgcz$metadata$coding
print(lookup.list$zone)
#>  [1] "BAFA"
#>  [2] "BG"
#>  [3] "BWBS"
#>  [4] "CDF"
#>  [5] "CMA"
#>  [6] "CWH"
#>  [7] "ESSF"
#>  [8] "ICH"
#>  [9] "IDF"
#> [10] "IMA"
#> [11] "MH"
#> [12] "MS"
#> [13] "PP"
#> [14] "SBPS"
#> [15] "SBS"
#> [16] "SWB"
#> [17] NA
```

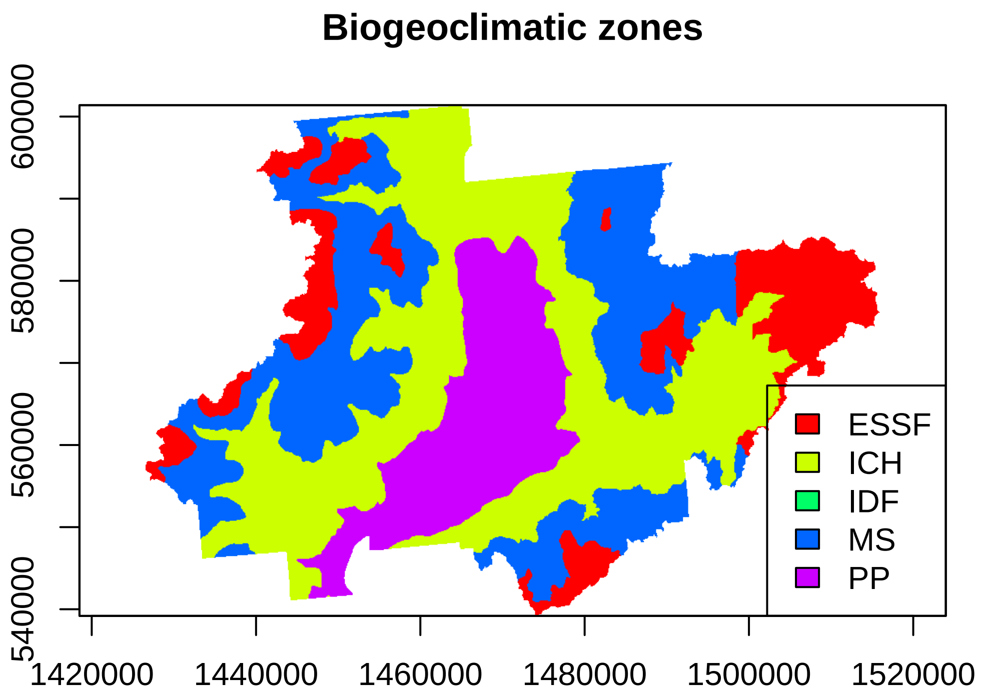
For example we have zone, 1 = Boreal Altai Fescue Alpine (BAFA), 2 = Bunchgrass, etc. See the documentation for the bgcz source script for links to a complete description of all codes.

The code below plots this data for the example region, replacing in the legend the integer levels of the raster with the properly matched zone codes:

```
# open the biogeoclimatic zone raster
bgcz.raster = getdata_bc(geo=example.sf, collection='bgcz', varname='zone', quiet=TRUE)
#> all 3 block(s) found in local data storage. Nothing to download

# a levels table (dataframe) can be extracted with `base::levels`
bgcz.levels = levels(bgcz.raster)[[1]]

# set up a colour palette and plot with legend defined manually
bgcz.levels$color = rainbow(nrow(bgcz.levels))
plot(bgcz.raster, legend=FALSE, col=bgcz.levels$color, main='Biogeoclimatic zones')
legend('bottomright', legend=bgcz.levels$code, fill=bgcz.levels$color)
```



Biogeoclimatic zones
600000
580000
560000
540000
1420000
1440000
1460000
1480000
1500000
1520000
🟥 ESSF
🟨 ICH
🟩 IDF
🟦 MS
🟪 PP