Tutorial to the R-package RCHILD (version 0.2.3)

This tutorial is of preliminary stage and mainly used for presenting and testing the capability of the (also still preliminary R-package RCHILD). RCHILD is a collection of functions that help to use the landscape evolution model CHILD (Channel-Hillslope Integrated Landscape Development) more conveniently. Original model output consists of a series of ASCII-files. To create maps, scenes and further thematic plots from this data, the use of functions is necessary. This document presents functions of the package and illustrates their argument usage.

To use the package, it needs to be installed and loaded. Installing from local source (yet, the package is not hosted on any server) can be done by `install.packages ("path_to/RCHILD_0.1.tar.gz", repos = NULL)`. The package relies on some other packages (`akima`, `fields`, `geometry`, `raster`, `rgdal`, `rgl`,`animation`) and additional programs and platform-dependent libraries. The package `raster` requires the GDAL-library (http://www.gdal.org/) to be installed. GDAL normally is already installed on Windows machines. For installation under Linux try in the terminal `sudo apt-get install proj` and `sudo apt-get install gdal-bin`. The package `animation` relies on other software to create animated gif-images (http://www.imagemagick.org/script/index.php) and avi-movies (http://ffmpeg.org/download.html).

Once installed, the package can be loaded by `library(RCHILD)`. When this is done successfully, all functions of the package are available.

2. The example data sets

2.3. DEM500

2.3.1. Overview

The example data set `DEM500` is a digital elevation model of the eastern part of the Erzgebirge and surrounding areas in Saxony, eastern Central Europe. It covers several geomorphologic and tectonic units. It is a `raster` data set, created from an ASTER GDEM data set, has a spatial resolution of 500 m and builds a grid with 139 columns and 143 rows. It has a geographic projection and is referenced in the UTM system, zone 33N. The data set can be loaded to R and its summary may be checked:

```
data(DEM500)  # load example data set
DEM500  # show summary of the DEM
## class       : RasterLayer
## dimensions  : 143, 139, 19877  (nrow, ncol, ncell)
## resolution  : 500, 500  (x, y)
## extent      : 361217, 430717, 5605429, 5676929  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## values      : in memory
## layer name  : erzgebirge_250m
## min value   : 92.92
## max value   : 915.3
```

2.3.2. Plot

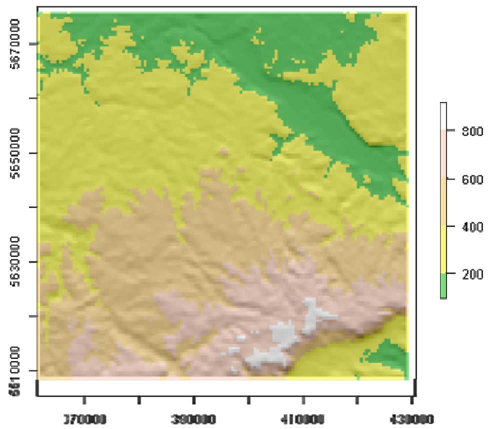
To plot the DEM there is a function `plot()` in the package `raster`, already loaded with the package `RCHILD`. So there would be nothing more to do than entering `plot(DEM500)`. However, to show a little bit more features of the package `raster` let us compute a hillshade to add some perspective visualisation. Hillshade data sets rely on information about slope and aspect. Both data sets can be computed and serve as input for the hillshade creation. However, the data set needs a spatial projection specified (see below for more information).

```
projection(DEM500)
## [1] "NA"
```

```
# set the projection to UTM zone 33N
projection(DEM500) <- "+proj=utm +zone=33 +north +ellps=WGS84 +datum=WGS84
+units=m +no_defs"

# calculate slope, aspect and, finally, the hillshade data set
slope     <- terrain(DEM500, opt = "slope")  # create slope data set
aspect    <- terrain(DEM500, opt = "aspect")  # create aspect data set
hillshade <- hillShade(slope, aspect, angle = 45, direction = 0)  # create
hillshade data set
```

Now all prerequisites are there to create an illustrative plot of the surface. This is done by first plotting the hillshade and then adding the DEM with some transparency. The hillshade must be in grey scale so the colour ramp is defined as a sequence of 250 values from 0 to 1. There should be no legend of the hillshade plotted. The DEM should be plotted in terrain colours, in this case classified to 5 classes. The degree of transparency is controlled with the argument alpha. To add the DEM plot to the allready existing hillshade, the argument add is set TRUE:

```
plot(hillshade, col = grey(seq(0, 1, length.out = 250)), legend = FALSE)
plot(DEM500, col = terrain.colors(5), alpha = 0.5, add = TRUE)
```



3. A typical model run and evaluation

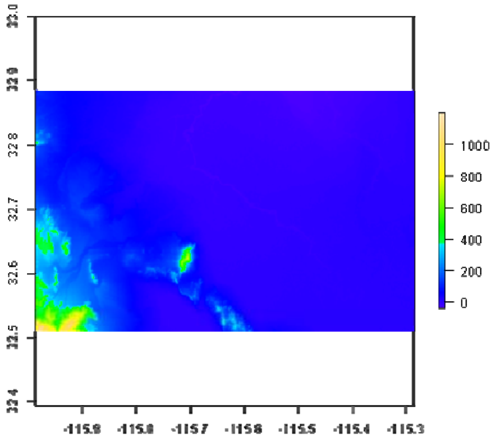
3.1. DEM preprocessing

3.1.1. Import a DEM raster-file

The packges `rgdal` and `raster` provide powerful import functions for different raster file types. In the packgage `RCHILD` these functions are used in the function `read.raster()`. It allows to read ESRI-files (asc), ERDAS Imagine-files (img) and GeoTiff-files (tif). Assume there is an ESRI-ASCII file (`LS_SRTM.asc`) in the current working directory, showing the north-eastern part of the Laguna Salada, Baja California, Mexico. This file may be imorted, querried and plotted by the following code:

```
library(raster)
LS.raw <- read.raster("LS_SRTM.asc")  # read the raster file
LS.raw  # show a brief summary of the imported data set
## class       : RasterLayer
## dimensions  : 188, 351, 65988  (nrow, ncol, ncell)
## resolution  : 0.002, 0.002  (x, y)
## extent      : -116, -115.3, 32.51, 32.89  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## values      : E:\Documents\Arbeit\Projects\LEM\R-LEM\LS_SRTM.asc
## layer name  : LS_SRTM
## min value   : -2.147e+09
```

```
## max value    : 2.147e+09
plot(LS.raw, col = topo.colors(250))  # plot the raster-object
```



3.1.2. Project the raster object

Depending on the source of the geospatial data the geographic projection and coordinate system may not be sufficiently appropriate for further handling. CHILD uses metric units. So it would be useful to have a raster data set in metric units as well. A typical projection and coordinate system is UTM (univeral transverse Mercator projection) and WGS84 (World Geodetic System 1984). If the data set does not already have a projection specified, it must be provided in R after importing the raster object. To check if a projection is defined the function `projection()`queries the raster object:

```
projection(LS.raw)
## [1] "NA"
```

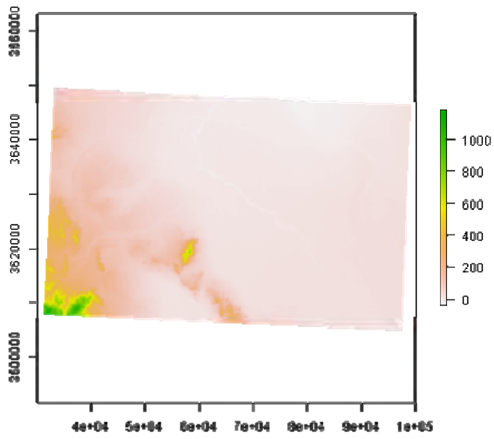
It returns NA, which means there is no projection present. Since we know that the data set is in a geographical coordinate system with decimal degrees as units (see axes units of the above plot) and the WGS84-system, this information can be set via the `projection()`-function.

```
projection(LS.raw) <- "+proj=longlat +ellps=WGS84 +towgs84=0,0,0,0,0,0,0
+no_defs "
```

This projection string looks a little bit weired. It contains all the information necessary for the PROJ4-library to handle projection, datum and ellipsoid. For more information google might be a good choice.

Since it is desireable to re-project the data set into a metric system, and UTM is a commonly used one, there is a function `projectRaster()` that does this job:

```
# specify the new projection, i.e. UTM zone 12 North with WGS84
newproj <- "+proj=utm +zone=12 +north +ellps=WGS84 +datum=WGS84 +units=m
+no_defs"

# re-project the raster object
LS.raw.proj <- projectRaster(from = LS.raw, crs = newproj)

# plot the re-projected raster object
plot(LS.raw.proj)
```

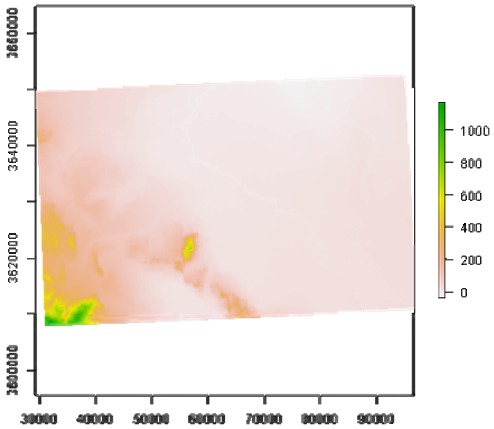


Note how i) the position of the data set in the plot changed due to the now projected character and ii) how the labels of the plot are now in metric units.

3.1.3. Rotate a raster object

CHILD allows using different types of tectonic forcing. Most of these require specifying a x- or y-coordinate which represents the location of a fault line. This in turn means that the DEM must be rotated in order to have the fault lines in either horizontal or vertical direction. Rotation of a raster object in R is a non-trivial job. So either the rotation must be performed by another software (e.g. QGIS, ArcGIS) prior to importing a data set into R or the following way must be chosen. In summary, the raster must be transformed to a Spatial-Points-data set, which can be rotated. Then the point cloud must be re-converted into a raster object again. To do all these tasks, a further package (`maptools`) is necessary and a function of the package `RCHILD` must be introduced.

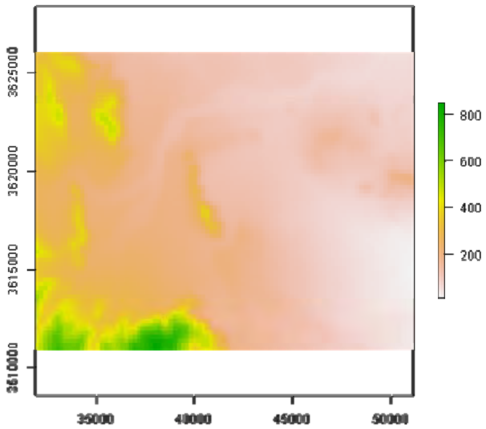
```
# 1. convert the raster object into a SpatialPointsDataFrame-object
LS.spdf <- SpatialPointsDataFrame(SpatialPoints(LS.raw.proj),
                                  proj4string = projection(LS.raw.proj),
                                  data = as.data.frame(values(LS.raw.proj)))
# 2. rotate the SpatialPointsDataFrame-object using the library maptools by
an angle of 340 degrees
library(maptools)
## Loading required package: foreign
## Loading required package: grid
## Loading required package: lattice
## Checking rgeos availability: FALSE Note: when rgeos is not available,
## polygon geometry computations in maptools depend on gpclib, which has a
## restricted licence. It is disabled by default; to enable gpclib, type
## gpclibPermit()
angle <- 355
LS.spdf.rot <- elide(LS.spdf, rotate = angle)

# 3. Extract the x-y-z-values of the rotated object
LS.TIN.rot <- cbind(LS.spdf.rot@coords,
as.numeric(LS.spdf.rot@data$values))
LS.TIN.rot <- LS.TIN.rot[!is.na(LS.TIN.rot[,3]),]

# 4. Interpolate the x-y-z-values
LS.rot <- TIN.raster(LS.TIN.rot, resolution = mean(res(LS.raw.proj)))

# 5. Plot the raster data set
plot(LS.rot)
```



3.1.4. Create a spatial subset (crop)

To work only with a spatial subset of the raster object, the data set can be cropped. The respective function `crop()` requires to specify a boundary box for the region to be returned. The extent of the boundary box, in turn, is created by the function `extent()`, which requires the following four values in exactly this order: xmin, xmax, ymin, ymax.

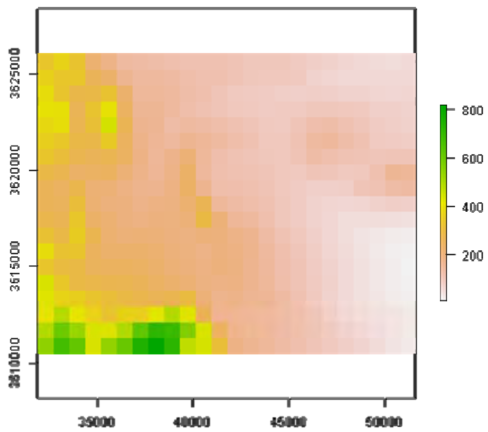
```
boundarybox <- extent(31800, 51200, 3611000, 3626000)
LS.crop <- crop(LS.rot, boundarybox)
plot(LS.crop)
```



3.1.5. Change the raster resolution

Changing the resolution may be done by (at least) two approaches: aggregating and resampling. Aggregating means that one new raster pixel will be computed from the aggregation of n original pixels (e.g. aggregation by 3 means that one new pixel will result from 3 by 3 original pixels). If this is desired, the function `aggregate()` will do the task. If a more flexible way is needed (e.g. not n multiples of the original pixel size but rather a new, arbitrary value) then the function `resample()` is needed. Since `resample()` requires a lot of further parameters to be set, it might be worthy to use `aggregate()` instead, if this function is sufficient.

```
LS.agg <- aggregate(LS.crop, fact = 4)
plot(LS.agg)
```



3.1.6. Create and write the CHILD PTS-file

CHILD can either create an elevation mesh from user-set parameters or use a PTS-file as input. A PTS-file defines the nodes of the mesh by providing x-y-z-coordinates and the boundary flags for each mesh node. PTS-files may of course be created purely by hand, i.e. by typing the coordinates and boundary flags, or by using an existing DEM to retrieve the elevation data from. The package `RCHILD` provides a function `create.PTS()` to do this job and to set the boundary flags as well.

There are several important parameters to discuss. Of course the function needs a **DEM data set** (`raster-object`), in the best case without NA-values since mesh nodes that are assigned NA-values from the DEM will be deleted from the output file. This is especially relevant for marginal nodes because if they become deleted, the boundary flags of the formerly interior nodes are not updated and CHILD might report a crash. So, *it is important to avoid NA-values throughout the extent of the CHILD mesh*.

The **extent** of the CHILD mesh determines how large the area will be that are modelled. The extent must be provided as a vector of length four, containing the bounding box in the order west, east, south, north. If there is no extent specified, the function will use the bounding box of the DEM to infer the mesh extent. As already noted, there should be no NA-values present in the DEM, especially not in the boundary regions of the mesh. However, since a raster data set is a matrix, DEM files that are rotated - even if only slightly, e.g. due to their geographic projection - usually contain NA-values at their margins.

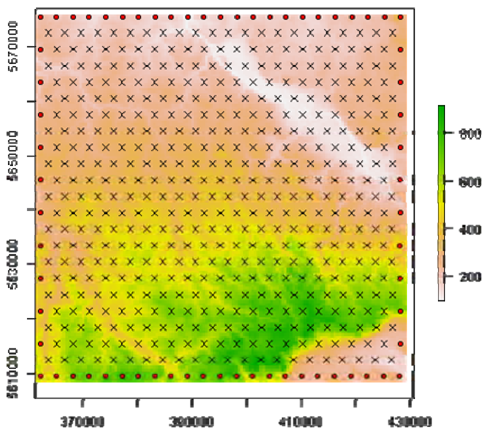
The **spacing** of the mesh is comparable to the resoulution of the DEM. However, since there are a couple of different mesh node arrangement types spacing is not directly equivalent to raster resolution. The mesh spacing should be in the range of the raster resolution but a little bit coarser. As mentioned in chapter 3.1.5, it is therefore useful to aggregate or resample the raster object prior to creating the PTS-file from it. Keep in mind that the node spacing is one of the key parameters controlling the time needed for a model run, so it is important to find the right way between statisical underestiamtion of the value space and loss in model performance.

There are several **types of node arrangement**, from completely random over partly random and raster to hexagonal. Furthermore, there is the possibility to create the PTS-file from user-defined x-y-coordinates. Depending on the research question these different types may yield different results. By default, the type is set to hexagonal.

Finally, there are some options of how to set the **boundary flags** of the PTS-file. The boundary type determines how CHILD will deal with sediment and discharge at respective boundary nodes. If a boundary is interior, there is always (and has to be by definition) a downstream neighbour to pass discharge and sediment on to. At boundary nodes, sediment and discharge may either meet a barrier (closed node) or an outlet of the model region (open boundary), along with respective consequences for the model result. Usually, when modelling catchment-wide areas of interest the easiest way is to choose a mesh larger than the catchment and to set all boundary flags to open, so the model will operate within the catchment by itself with no interest put on the catchment-external boundary conditions. So *the default boundary flag option is "open"*. However, there may be situations that require closing the entire mesh (argument value "closed") or to let the function find the marginal node with the lowest elevation and set its flag to open (argument value "lowest").

Enough for the background. To create a PTS-file for the landscape described by DEM500, with a node spacing of 3000 m and hexagonal mesh geometry, a little bit smaller than the original DEM and with open boundaries the following code will serve. Furthermore, a little bit more code is provided, illustrating how to visualise the results:

```
# set the mesh extent vector according to the DEM extent with a margin of
# 1000 m
mesh_extent <- c(DEM500@extent@xmin + 1000, DEM500@extent@xmax - 1000,
DEM500@extent@ymin +
    1000, DEM500@extent@ymax - 1000)

# create the PTS-object and further data
PTS3000 <- create.PTS(DEM = DEM500, extent = mesh_extent, spacing = 3000,
type = "hexagonal",
    boundary = "open")

# create another PTS-object, see below for use
PTS1000 <- create.PTS(DEM = DEM500, extent = mesh_extent, spacing = 1000,
type = "hexagonal",
    boundary = "open")

nodes_interiour <- PTS3000[PTS3000[, 4] == 0, ]  # all interiour nodes
nodes_boundary <- PTS3000[PTS3000[, 4] == 2, ]  # all boundary nodes

# visualise the result
plot(DEM500)  # plot DEM
points(nodes_interiour[, 1:2], pch = 4)  # plot interiour nodes
points(nodes_boundary[, 1:2], pch = 21, bg = "red")  # plot boundary nodes
```



To finish this chapter, the PTS-object needs to be saved to a real ASCII-file. This is done by the function `write.PTS()`, requiring an output filename and the name of the PST-object:

```
write.PTS(filename = "PTS3000.pts", PTS = PTS3000)
```

3.2. Run CHILD and import the results

To run CHILD one needs at least two files: a compiled version of CHILD (assumed to be present in the current working directory if not specified different by the argument `path`) and an input-file (i.e. a text file containing all necessary model control parameters). Most problems arise from inappropriately created input-files. There are more than 230 parameters available. To aid creating a sound input-file, the package `RCHILD` offers some functions. The basic steps when starting with nothing are: i) create a new input-file object, ii) add all necessary parameters, iii) check the object for correctness and consistency, iv) write the actual *.in-file, v) run CHILD with this input-file.

3.2.1. Create or read and manipulate the input-file

If there is no input-file template present that may be modified to use it, one has to create a completely new file and specify the necessary parameters. To facilitaty this step, the function `create.IN()` can be used. It creates an empty S4-object `IN`. This object can then be filled with appropriate parameters:

```
# create an empty input-file
test.in <- create.IN()

# show some general (but empty) parameters
test.in@OUTFILENAME
## [1] NA
test.in@RUNTIME
## [1] NA
test.in@UPTYPE
## [1] NA

# set some parameters exemplary
test.in@OUTFILENAME <- "testoutput"
test.in@RUNTIME <- 1000

# show the updated parameters
test.in@OUTFILENAME
## [1] "testoutput"
test.in@RUNTIME
## [1] 1000
```

All empty slots of the IN-object are assigned `NA` by default. This means they will be deleted from the input-file during the writing process, performed with `write.IN()`. So, if the user manipulates an input-file, either in its text form or as an S4-object, empty parameters should be assigned `NA`, too.

Since it is unlikely that a user has all necessary parameters and their interdependencies in mind and wants to create a new input-file each time a new model shall be run it is meaningful to use an already existing input-file (e.g. from a previous run) and modify it.

Assuming there is an inputfile `erzgebirge_open.in` present in the working directory, this can be read to an S4-object `IN` with the function `read.IN()`.

```
# read the input file and show some general parameter settings
erzgebirge_open.in <- read.IN("erzgebirge_open.in")
erzgebirge_open.in@OUTFILENAME  # show output file name
## [1] NA
erzgebirge_open.in@RUNTIME  # show model run time
## [1] NA
erzgebirge_open.in@POINTFILENAME  # show PTS file name
## [1] "PTS1000.pts"
```

Usually it is desired to change or modify the input-file parameters and check the result. As shown several times before, this can be done easily. Doubling the model run time can be done like this:

```
erzgebirge_open.in@RUNTIME <- erzgebirge_open.in@RUNTIME * 2
```

If the correctness of the input-file is no matter of debate, the (modified) S4-object can be converted (back) to an input-file in ASCII format (`*.in`) with the function `write.IN()`:

```
write.IN(erzgebirge_open.in, filename = "erzgebirge_open2.in")
```

However, usually a test of the input-file is useful. Therefore, a function `check.IN()` can be used. Currently, the function only checks if all mandatory model parameters are present in the S4-object. In the future it will also check for redundant or unused parameters or even if important parameter settings are within a meaningful value range. The function returns a list object with the potentially modified IN-file and all found warnings and notifications.

The function can operate in active and passive mode (passive mode is the default option). In active mode it automatically corrects wrongly specified or missing parameters to at least ensure that CHILD will run. However, this active mode should be used with care since parameters will be set to a default value, probably unsuitable to the specific modelling task. This mode is primarily there to be used in a loop where active adaption is necessary. It is therefore recommended to use the passive mode, i.e. warnings and notifications are recorded and returned. This way, the user can decide personally which parameters need adjustment.

To simply test which parameters are mandatory to an input-file, in the following code an empty IN-object is created and then checked:

```
empty.in <- create.IN()  # create empty IN-object
empty.in.checked <- check.IN(empty.in)  # check the object
##        [,1]
##   [1,] "Warning: parameter OUFILENAME missing."
##   [2,] "Warning: parameter RUNTIME missing."
##   [3,] "Warning: parameter OPINTRVL missing."
##   [4,] "Warning: parameter OPTREADINPUT missing."
##   [5,] "Warning: parameter OPTVAR missing."
##   [6,] "Warning: parameter ST_PMEAN missing."
##   [7,] "Warning: parameter ST_STDUR missing."
##   [8,] "Warning: parameter ST_ISTDUR missing."
##   [9,] "Warning: parameter FLOWGEN missing."
##  [10,] "Warning: parameter OPTMEANDER missing."
##  [11,] "Warning: parameter OPTMNDR missing."
##  [12,] "Warning: parameter OPTDETACHLIM missing."
##  [13,] "Warning: parameter OPTREADLAYER missing."
##  [14,] "Warning: parameter OPTLAYEROUTPUT missing."
##  [15,] "Warning: parameter OPTTSOUTPUT missing."
##  [16,] "Warning: parameter OPTINTERPLAYER missing."
##  [17,] "Warning: parameter OPTSTRATGRID missing."
##  [18,] "Warning: parameter OPTFLOODPLAIN missing."
##  [19,] "Warning: parameter OPTLOESSDEP missing."
##  [20,] "Warning: parameter OPTEXPOSURETIME missing."
##  [21,] "Warning: parameter OPTVEG missing."
##  [22,] "Warning: parameter OPTKINWAVE missing."
##  [23,] "Warning: parameter DETACHMENT_LAW missing."
##  [24,] "Warning: parameter MB missing."
##  [25,] "Warning: parameter NB missing."
##  [26,] "Warning: parameter PB missing."
##  [27,] "Warning: parameter TAUCB missing."
##  [28,] "Warning: parameter TAUCR missing."
##  [29,] "Warning: parameter TRANSPORT_LAW missing."
##  [30,] "Warning: parameter KD missing."
##  [31,] "Warning: parameter DIFFUSIONTHRESHOLD missing."
##  [32,] "Warning: parameter OPTDIFFDEP missing."
##  [33,] "Warning: parameter BEDROCKDEPTH missing."
##  [34,] "Warning: parameter REGINIT missing."
##  [35,] "Warning: parameter MAXREGDEPTH missing."
##  [36,] "Warning: parameter UPTYPE missing."
##  [37,] "Warning: parameter NUMGRNSIZE missing."
##  [38,] "Warning: parameter GRAINDIAM1 missing."
##  [39,] "Warning: parameter BRPROPORTION1 missing."
##  [40,] "Warning: parameter REGPROPORTION1 missing."
##  [41,] "Warning: parameter CHAN_GEOM_MODEL missing."
##  [42,] "Warning: parameter HYDR_WID_COEFF_DS missing."
##  [43,] "Warning: parameter HYDR_WID_EXP_DS missing."
##  [44,] "Warning: parameter HYDR_WID_EXP_STN missing."
##  [45,] "Warning: parameter HYDR_DEP_COEFF_DS missing."
```

```
## [46,] "Warning: parameter HYDR_DEP_EXP_DS missing."
## [47,] "Warning: parameter HYDR_DEP_EXP_STN missing."
## [48,] "Warning: parameter HYDR_ROUGH_COEFF_DS missing."
## [49,] "Warning: parameter HYDR_ROUGH_EXP_DS missing."
## [50,] "Warning: parameter HYDR_ROUGH_EXP_STN missing."
## [51,] "Warning: parameter BANKFULLEVENT missing."
## [52,] "no notifications, congratulations!"
```

A check of the modified real input-file `erzgebirge_open.in` would yield the following results:

```
IN.checked <- check.IN(erzgebirge_open.in)
## [1,] "no warnings, congratulations!"
## [2,] "no notifications, congratulations!"
```

There should be no warnings or notifications found. Hence, it should be possible to write the input-file as described above and to run CHILD.

3.2.2. Run CHILD

The function `run.CHILD()` requires the file name of an input-file (with extension) and, if not present in the current working directory, the path to a compiled version of CHILD. Usually, CHILD is run from the command line but R offers calling and tracing system commands in the console via the function `system`. Although this is no real implementation of CHILD into R it offers at least the chance to follow the CHILD run information on the screen. After the model is (hopefully) finished, the output files are read to an S4-object CHILD and the created output files are removed from the working directory, if this is not disabled by setting the parameter `outfiles` to `TRUE`.

To model landscape evolution of the wider eastern Erzgebirge region for 8000 years, using a hexagonal PTS file with 1000 m node spacing (remember the section above), the following code should serve. BUT REMEMBER, a CHILD model run may take considerable time:

```
# write PTS file
write.PTS(filename = "PTS1000.pts", PTS1000)

# update PTS file name in IN-object
erzgebirge_open.in@POINTFILENAME <- "PTS1000.pts"

# write updated input file
write.IN(erzgebirge_open.in, filename = "erzgebirge_open.in")

# check updated input file (again)
check2 <- check.IN(erzgebirge_open.in)
##        [,1]
## [1,] "no warnings, congratulations!"
## [2,] "no notifications, congratulations!"

# run the model ERZ <- run.CHILD('erzgebirge_open.in')
load("ERZ.RData")
```

3.2.3. Read the model results to R

When using the function `run.CHILD()`, the results will be automatically imported to an S4-object `CHILD` and written to a specified variable. However, it is also possible to import output files of an external CHILD-run to a CHILD-object. Even the function `run.CHILD()` allows going this way. Usually, the output files of the model run will be deleted after import into the S4-object but setting the logical parameter outfiles from `FALSE` to `TRUE` allows keeping the individual files. Importing external CHILD-run output files is done by the function `read.CHILD()`.

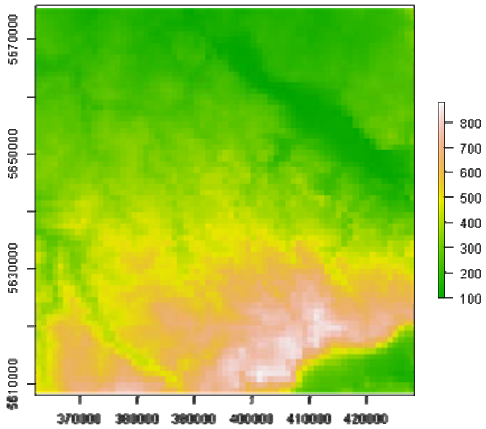
3.3. Graphical visualisation of the results

The great advantage of CHILD is that it runs fast, a not so minor disadvantage is that the entire model output consists of numerical data, only.

3.3.1. Display the surface and additional elements

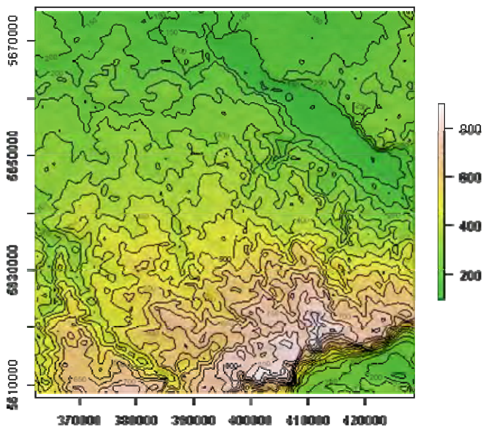
The easiest and quickest way to generate a visual impression of a model run output is to plot a map-like image of the colour-coded elevation data. Of course, the time step for which the plot will be created must be specified, but nothing more: just a “quick-and-dirty” visualisation can be created by:

```
display.surface(ERZ, 1)
```



To add some illustrative features, one may think of adding a hillshade layer and/or a contours layer. Remembering the things explained about creating hillshades from raster objects (chapter 2.3.2) it is obvious that the data needs a projection to be specified. Vertical exaggeration increases the visual effect of the hillshade. Contour lines also add some more impression of the relief. Furthermore, by default the resolution of the image is adjusted to the mean node spacing of the mesh. Interpolation to a finer raster size is possible. So a little bit more finished map would be possible by running the following code:

```
display.surface(dataset = ERZ, timestep = 1, resolution = 300, projection =
"+proj=lcc +lat_1=48 +lat_2=33 +lon_0=-100 +ellps=WGS84",
    hillshade = TRUE, exaggeration = 5, contours = TRUE)
```



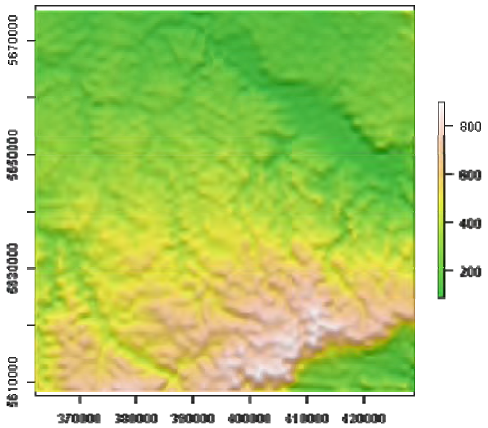
Of course all the visualisations, created by `display.surface()` can also be created by explicit code. Usually, this includes creating a raster object of the modelled surface, most easily by the function `TIN.raster()`, which in turn requires a TIN first:

```
ERZ_01_TIN <- read.TIN(dataset = ERZ, timestep = 5)
ERZ_01_raster <- TIN.raster(TIN = ERZ_01_TIN, resolution = 300, method =
"tps")
```

`TIN.raster()` supports different interpolation techniques for the irregular spaced TIN-data: “lin” (linear interpolation, the default method), “cub” (cubic spline interpolation) and “tps” (thin plate spline interpolation).

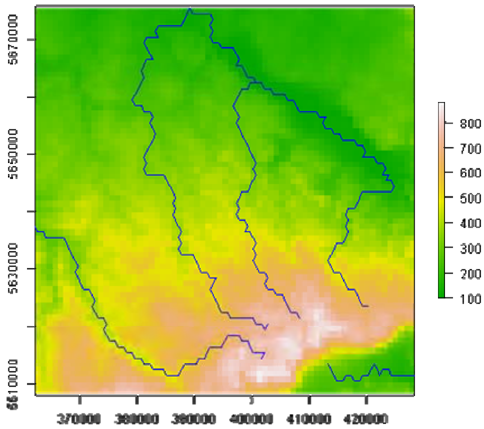
The raster object can be plotted and - given it has a geographical projection - derivatives such as a hillshade can be created as well (see above):

```
projection(ERZ_01_raster) <- "+proj=lcc +lat_1=48 +lat_2=33 +lon_0=-100
+ellps=WGS84"
hillshade <- hillShade(terrain(ERZ_01_raster * 10, opt = "slope"),
terrain(ERZ_01_raster *
    10, opt = "aspect"), angle = 45, direction = 0)
plot(hillshade, col = grey(seq(0, 1, length.out = 250)), legend = FALSE)
plot(ERZ_01_raster, col = terrain.colors(250), alpha = 0.7, add = TRUE)
```



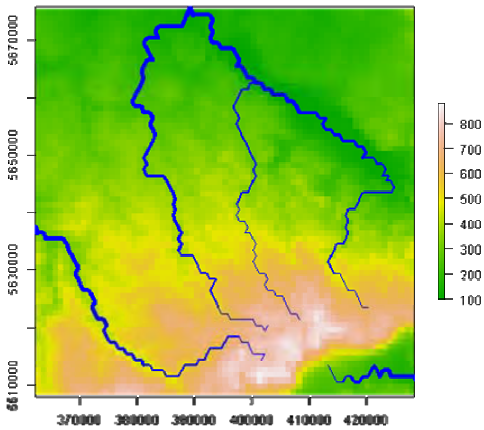
Beyond plotting the modelled relief as a map it would be useful to add streams to the visual output. The function `display.surface()` allows specifying start points for streams and displays the traced streams as blue lines on top of the map. There is also a function `click.coordinates()` which allows to simply click on a map and retrieve the coordinates rather than figuring the positions out by trial and error or using an external GIS. R-code may look like `springs <- click.coordinates(5)`. These springs can be specified in the function `display.surface()`:

```
springs <- cbind(c(420129.4, 408496.2, 402948, 402053.2, 413000.8),
c(5623119, 5621150, 5620077, 5614528, 5613000))
display.surface(ERZ, 1, stream = TRUE, startpoints = springs)
```



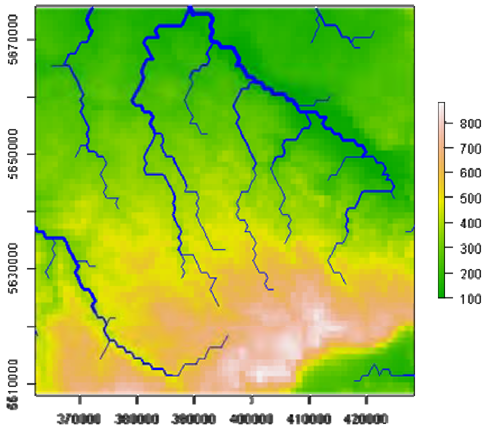
Instead of plotting streams as lines of fixed width, it is possible to scale line width according to contribution area. This in done by specifying two additional parameters: `width_scale` and `width_max`. The first one determines how line width should be scaled. From `"none"` (no scaling, default value) over `linear` (linear scaling), `root`(square root scaling) and `power` (power scaling) to `log` (logarithmic scaling), there are plenty of types to choose from. The second parameter determines the maximum possible line width (i.e. that part of the streams with the largest contribution area). As an example, the same five streams as defined above are displayed with root-scaled line widths up to 5 units by the following code:

```
display.surface(ERZ, 1, stream = TRUE, startpoints = springs, width_max =
                5, width_scale = "root")
```



Furthermore, it might be illustrative and necessary to display not only selected streams but the entire stream network of a modelled landscape. This can be done by setting the parameter network to TRUE. Additionally, a threshold contribution area (`area_min`) may be specified upon which a stream is defined. In the example landscape a meaningful threshold would be around 50 million square metres:

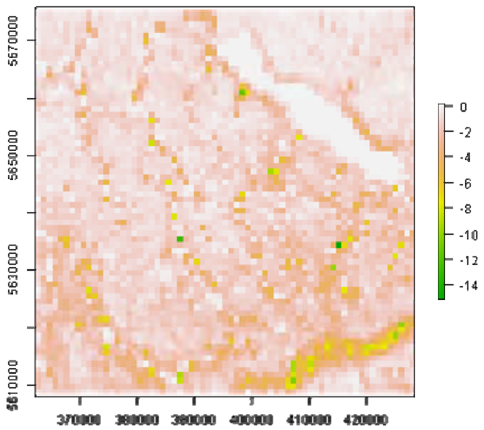
```
display.surface(ERZ, 1, network = TRUE, area_min = 5 * 10^7, width_max = 5,
                width_scale = "root")
```



Behind the described display options regarding streams and stream networks there are two functions that allow tracing streams and stream networks individually, with additional numeric output: `trace.stream()` and `trace.network()`.

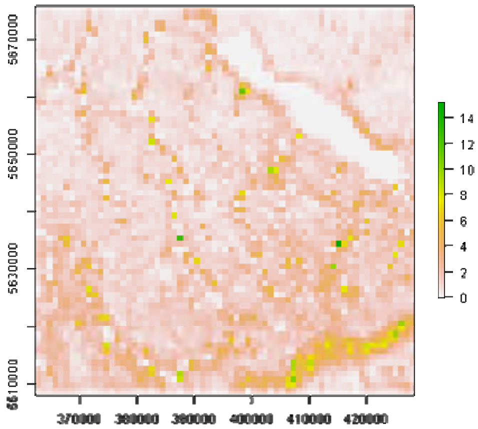
Apart from plotting elevation it is also possible to plot differences in elevation between two time steps. The function `display.surface()` supports this by specifying two time steps instead of just one. Then, the difference (i.e. second value minus first value in the `timestep` vector) is plotted:

```
display.surface(ERZ, timestep = c(2, 1))
```



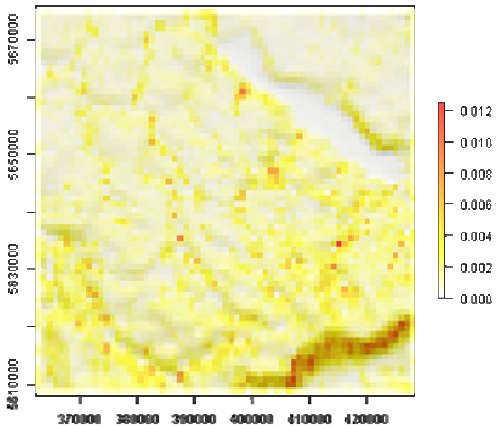
A similar result would be possible by explicit calculation, as well:

```
DEM1 <- TIN.raster(read.TIN(ERZ, 2))
DEM2 <- TIN.raster(read.TIN(ERZ, 1))
DEMdiff <- DEM2 - DEM1
plot(DEMdiff)
```



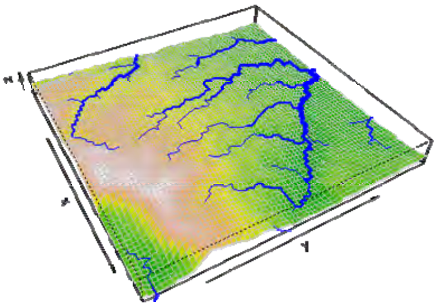
The difference between elevations of two time steps is close to a mean erosion rate. However, the latter requires including tectonic uplift and dividing by the time period that separates the two data sets. Mean erosion rates can be displayed by setting the function parameter `erosion` to `TRUE` and by specifying a value for tectonic uplift (if there is uplift present) other than the value already present in the CHILD object (check `ERZ@inputs`). If only one time step is specified then its precursor is used by default, if two times steps (actual time step and preceding time step) are used, the function shows the mean erosion retain metres per year between these two time steps. Note that the hillshade and exaggeration parameters only affect the relief data and not the thematic layer (erosion or elevation difference).

```
display.surface(ERZ, timestep = c(5, 1), erosion = TRUE, hillshade = TRUE,
exaggeration = 10)
```



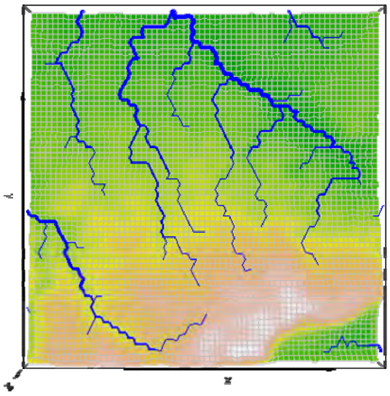
Although a map view is the most common type of surface visualisation, in some cases it may be better to have a perspective view. This can be done by setting the parameter `type` from `"map"` (the default value) to `"wireframe"`. This visualisation type supports all the previously described features (streams, stream networks, height differences, erosion rates) and uses two additional parameters for adjusting the perspective view: the azimutal angle `theta` and the height angle, i.e. the colatitude `phi`. Both are by default set to 30 °.

```
display.surface(ERZ, 1, network = TRUE, area_min = 5 * 10^7, width_max = 5,
    width_scale = "root", type = "wireframe", theta = 60, phi = 40)
```



A view close to a map view would be possible (although not meaningful) by setting `theta` to 0 and `phi` to 90):

```
display.surface(ERZ, 1, network = TRUE, area_min = 5 * 10^7, width_max = 5,
    width_scale = "root", type = "wireframe", theta = 0, phi = 90)
```



Yet another type of visualisation is rendering the landscape model as real time 3D scene, using openGL (the technique also used in Google Earth). To do so, `type` must be set to `"scene"`. However, up to now it is not yet possible to display streams and stream networks with this visualisation type:

```
display.surface(ERZ, 1, network = TRUE, area_min = 5 * 10^7, width_max = 5,
    width_scale = "root", type = "scene", exaggeration = 5)
```