10

Stream networks (baranjahill)

10.1 Introduction

The purpose of this exercise is to generate a map of stream networks using **error propagation**[1] techniques (Heuvelink, 2002), i.e. to generate multiple DEMs using conditional simulations, then derive a stream network for each realization, and finally evaluate how the propagated uncertainty correlates to various topographic parameters. Upon completion of this exercise, you will be able to generate loops, run SAGA via the R command line, and import and export raster maps from R to SAGA GIS and Google Earth. The last section in this exercise focuses on GRASS GIS and how to run a similar type of analysis in this open source GIS.

The "Baranja hill" study area, located in eastern Croatia, has been mapped extensively over the years and several GIS layers are available at various scales (Hengl and Reuter, 2008). The study area corresponds approximately to the size of a single 1:20,000 aerial photo. Its main geomorphic features include hill summits and shoulders, eroded slopes of small valleys, valley bottoms, a large abandoned river channel, and river terraces. Elevation of the area ranges from 80 to 240 m with an average of 157.6 m and a standard deviation of 44.3 m. The complete "Baranja hill" data set is available from the geomorphometry website[2].

In principle, the only input for this exercise is a point map showing field-measured elevations (ESRI Shapefile). This map will be used to generate multiple realizations of a Digital Elevation Model, and then extract drainage networks, as implemented in the SAGA GIS package. This exercise is based on previous articles published in Computers and Geosciences (Hengl et al., 2008). A similar exercise can be found in Temme et al. (2008). A detailed description of the RSAGA package can be can be found in Brenning (2008).

10.2 Data download and import

First, open a new R session and change the working directory to where all the data sets are located. Download the R script (`baranjahill.R`) needed to complete this exercise, and open it in some script editor. Before you start any processing, you will need to load the following packages:

```
> library(maptools)
> library(gstat)
> library(geoR)
> library(rgdal)
> library(lattice)
> library(RSAGA)
```

Check `rsaga.env` to make sure that RSAGA can find your local installation of SAGA. As indicated in §3.1.2, SAGA is not an R package, but an external application and needs to be installed separately.

We also need to set-up the correct coordinate system for this study area, which is the Gauss-Krueger-based coordinate system zone 6, with a 7–parameter datum definition:

[1] A practical guide to error propagation is available via `http://spatial-accuracy.org/workshopSUP`.

[2] `http://geomorphometry.org/content/baranja-hill`

```
# set the correct coordinate system:
> gk_6 <- "+proj=tmerc +lat_0=0 +lon_0=18 +k=0.9999 +x_0=6500000 +y_0=0
+           +ellps=bessel +units=m
+           +towgs84=550.499,164.116,475.142,5.80967,2.07902,-11.62386,0.99999445824
```

Next, download the shapefile (`elevations.shp`) and extract it to the local folder[3]:

```
> download.file("http://spatial-analyst.net/book/system/files/elevations.zip",
+     destfile=paste(getwd(),"elevations.zip",sep="/"))

  trying URL 'http://spatial-analyst.net/book/system/files/elevations.zip'
  Content type 'application/x-zip-compressed' length 165060 bytes (161 Kb)
  opened URL
  downloaded 161 Kb


> for(j in list(".shp", ".shx", ".dbf")){
>   fname <- zip.file.extract(file=paste("elevations", j, sep=""),
+        zipname="elevations.zip")
>   file.copy(fname, paste("./elevations", j, sep=""), overwrite=TRUE)
> }
> unlink("elevations.zip")
> list.files(getwd(), recursive=T, full=F)

  [1] "baranjahill.R"  "elevations.dbf"
  [3] "elevations.shp" "elevations.shx"
```

We can import the sampled elevations from `elevations.shp` to R using:

```
> elevations <- readShapePoints("elevations.shp", proj4string=CRS(gk_6))
> str(elevations)

  Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
    ..@ data       :'data.frame': 6367 obs. of  1 variable:
    .. ..$ VALUE: num [1:6367] 206 208 207 204 203 ...
    .. ..- attr(*, "data_types")= chr "N"
    ..@ coords.nrs : num(0)
    ..@ coords     : num [1:6367, 1:2] 6551880 6552027 6551949 6552134 6551846 ...
    .. ..- attr(*, "dimnames")=List of 2
    .. .. ..$ : chr [1:6367] "0" "1" "2" "3" ...
    .. .. ..$ : chr [1:2] "coords.x1" "coords.x2"
    ..@ bbox       : num [1:2, 1:2] 6551799 5070471 6555640 5074356
    .. ..- attr(*, "dimnames")=List of 2
    .. .. ..$ : chr [1:2] "coords.x1" "coords.x2"
    .. .. ..$ : chr [1:2] "min" "max"
    ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
    .. .. ..@ projargs: chr " +proj=tmerc +lat_0=0 +lon_0=18 +k=0.9999 +x_0=6500000
    +y_0=0 +ellps=bessel +units=m +towgs84=550.499,164.116,475.142,5.80967,2"|
    __truncated__


> names(elevations@data) <- "Z"
```

This shows that the data set consists of 6367 points of field measured heights. The heights can be used to generate a Digital Elevation Model (DEM), that can then be used to extract a stream network. For example, you can open the point layer in SAGA GIS, then use the module *Grid* ↦ *Gridding* ↦ *Spline interpolation* ↦ *Thin Plate Splines (local)* and generate a smooth DEM[4]. Then, you can preprocess the DEM to remove spurious sinks using the method of Planchon and Darboux (2001). Select *Terrain Analysis Preprocessing Fill sinks*, and then set the minimum slope parameter to 0.1. Now that we have prepared a DEM, we can derive stream networks using the *Channel Network* function which is available in SAGA under *Terrain Analysis* ↦ *Channels*. You can use e.g. 40 (pixels) as the minimum length of streams. This would produce a map (vector layer) as

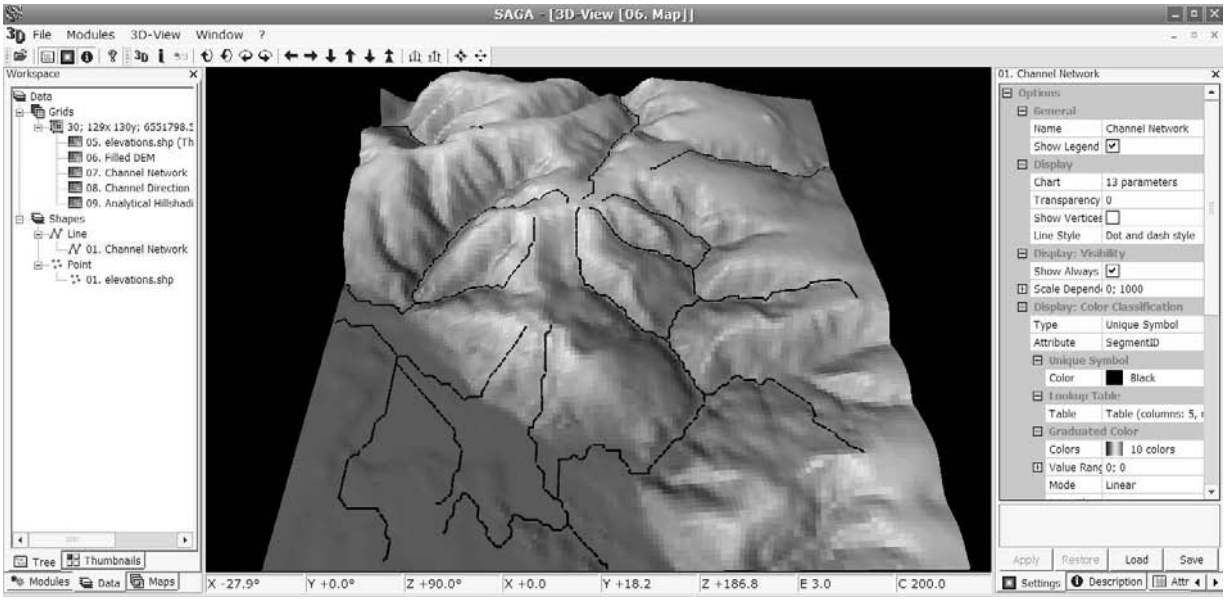

Fig. 10.1: Stream network generated in SAGA GIS. Viewed from the West side.

shown in Fig. 10.1. Assuming that the DEM and the stream extraction model are absolutely accurate, i.e. that they perfectly fit the reality, this would then be the end product of the analysis.

However, in reality, we know that errors exist — they are inherent both in measurements of elevations and in the stream extraction algorithm — and that they possibly have a significant influence on the final product (stream network). But how important is the influence of errors, and how are the errors connected with the geomorphometric properties of terrain? This is exactly what we will try to answer in this exercise.

10.3 Geostatistical analysis of elevations

10.3.1 Variogram modelling

In the previous exercise we generated a smooth DEM by using spline interpolation and by setting some parameters *'by hand'*. We would now like to produce a surface model using a more objective approach. We can take a step back and do some preliminary analysis in geoR to estimate possible anisotropy and evaluate how smooth is the elevation surface. First, let us look at the distribution of values:

```
> range(elevations$Z)
```

```
  [1]  85.0 244.2
```

```
# sub-sample -- geoR cannot deal with large data sets!
> sel <- runif(length(elevations@data[[1]]))<0.2
> Z.geo <- as.geodata(elevations[sel,"Z"])
# histogram:
> plot(Z.geo, qt.col=grey(runif(4)))
```

which shows that the values of Z are approximately normally distributed, with some clustering around low values (Fig. 10.2, bottom right). You can notice from Fig. 10.1 that the clustered low values are elevations measured in the floodplain.

To get some idea about the smoothness of the target variable and possible anisotropy, we can plot the two standard variograms:

[3]This exercise starts with a single input data set, but then results in a long list of maps! Make sure you have enough space on your computer.

[4]You can set 500 m as the search radius and grid resolution of 30 m.

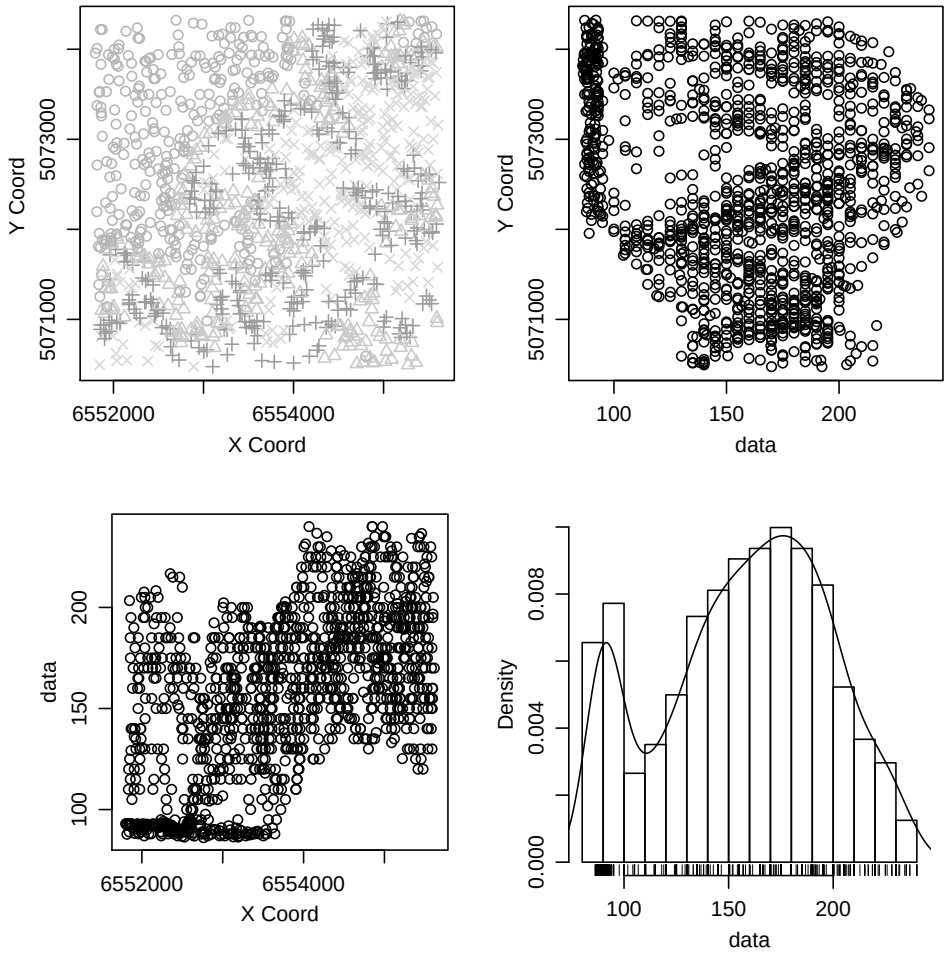

Fig. 10.2: Distribution of the target variable in geographical and feature space. A standard plot (`plot.geodata`) in geoR using a 20% sub-sample of the original data set.

```
> par(mfrow=c(1,2))
# anisotropy:
> plot(variog4(Z.geo, max.dist=1000, messages=FALSE), lwd=2)
# fit variogram using likfit:
> Z.svar <- variog(Z.geo, max.dist=1000, messages=FALSE)
# WLS fitting:
> Z.vgm <- variofit(Z.svar, ini=c(var(Z.geo$data), 1000), fix.nugget=T, nugget=0)
> Z.vgm

  variofit: model parameters estimated by WLS (weighted least squares):
  covariance model is: matern with fixed kappa = 0.5 (exponential)
  fixed value for tausq =  0
  parameter estimates:
    sigmasq       phi
  1352.3685  650.9268
  Practical Range with cor=0.05 for asymptotic range: 1950.002

  variofit: minimised weighted sum of squares = 31777661


> env.model <- variog.model.env(Z.geo, obj.var=Z.svar, model=Z.vgm)
> plot(Z.svar, envelope=env.model); lines(Z.vgm, lwd=2);
> dev.off()
```

which shows that the target variable (Z) varies equally in all directions, i.e. it can be modeled using isotropic models (Fig. 10.3). It is also a relatively smooth variable — there is no nugget variation and spatial autocorrelation is valid (practical range) up to a distance of 2 km. This is in fact a typical variogram for elevation data i.e. representation of a land surface. Note also that the confidence bands (envelopes) are now much narrower than in Fig. 5.15, possibly because there are more points and because the feature is more smooth.

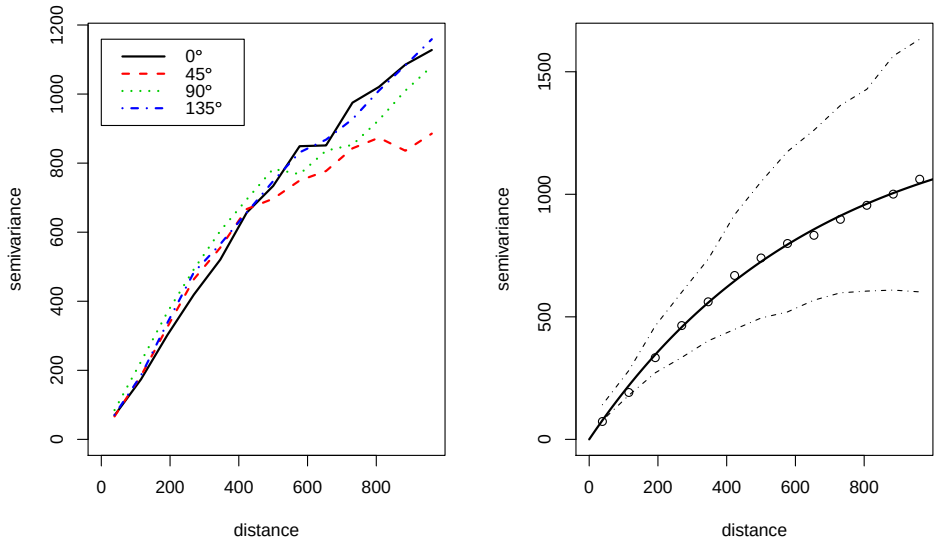

Fig. 10.3: Standard variograms fitted for elevations: (left) anisotropy in four directions; (right) isotropic variogram model fitted using the weighted least squares (WLS) and its confidence bands.

10.3.2 Geostatistical simulations

We can now use the variogram model to generate multiple realizations of the target variable. We first need to create a new empty grid for which a DEM can be derived. An empty grid with full topology can be generated in sp package:

```
> demgrid <- spsample(elevations, type="regular", cellsize=c(30,30))
> gridded(demgrid) <- TRUE
> fullgrid(demgrid) <- TRUE
> demgrid@grid
```

```
                        x1      x2
cellcentre.offset 6551822 5070484
cellsize               30      30
cells.dim             128     130
```

Now that we have fitted the variogram model and prepared a grid of interest, we can simulate *N* DEMs. We will use the **Stochastic Conditional Gaussian Simulations** algorithm as implemented in the gstat[5] package. The number of realizations *N* must be sufficiently large to obtain stable results, but exactly how large *N* should be depends on how accurate the results of the uncertainty analysis should be. The accuracy of the Monte-Carlo method is proportional to the square root of the number of runs *N*. Therefore, to double the accuracy one must quadruple the number of runs (Temme et al., 2008). This means that although many runs may be needed to reach stable and accurate results, any degree of precision can be reached by taking a large enough sample *N*. Consequently, the Monte-Carlo method is computationally demanding, particularly when the GIS operation

[5] Conditional simulations can also be generated in geoR, but gstat is much less time-consuming.

takes significant computing time (Heuvelink, 2002). As a rule of thumb, we will take 100 simulations as large enough.

Because we further use gstat, we need to copy the fitted variogram values to a vgm object:

```
# copy the values fitted in geoR:
> Z.ovgm <- vgm(psill=Z.vgm$cov.pars[1], model="Mat",
+      range=Z.vgm$cov.pars[2], nugget=Z.vgm$nugget, kappa=1.2)
> Z.ovgm

  model    psill    range kappa
1   Nug    0.000   0.0000     0
2   Mat 1352.368 650.9268   1.2
```

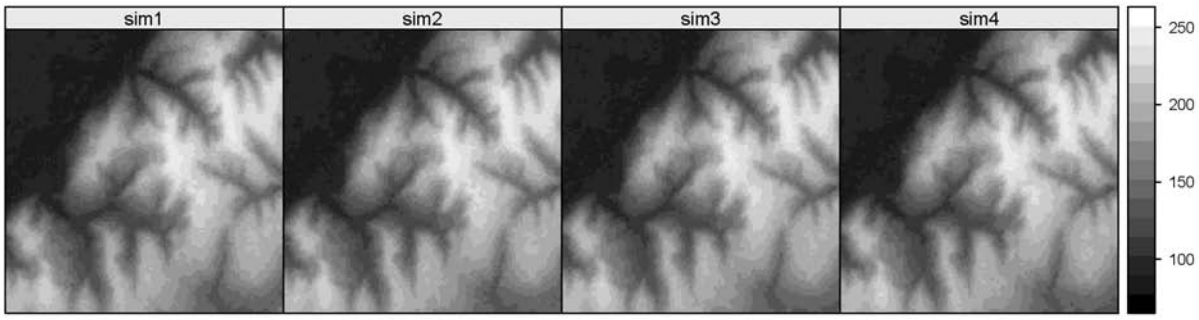

Fig. 10.4: Four realizations of the DEM following conditional geostatistical simulations.

Note that we have purposively set the kappa[6] parameter at 1.2 (it was originally 0.5). Following our knowledge about the feature of interest, we know that a land surface is inherently smooth — due to the erosional processes and permanent leveling of topography — so we wish to generate realizations of DEMs that fit our knowledge of the area.

The conditional simulations in gstat can be run by simply adding the nsim argument to the generic krige method:

```
> N.sim <- 100
> DEM.sim <- krige(Z ~ 1, elevations, demgrid, Z.ovgm, nmax=30, nsim=N.sim)

  drawing 100 GLS realisations of beta...


# this can take few minutes!!
> fullgrid(DEM.sim) <- TRUE
> spplot(DEM.sim[1:4], col.regions=grey(seq(0,1,0.025)))
```

which shows that we have accomplished our objective: we have managed to simulate 100 equiprobable DEMs that are equally smooth as the DEM shown in Fig. 10.1. To visualize the differences between realizations, we can make a cross section and plot all simulated surfaces on top of each other (Fig. 10.5):

```
# Cross-section at y=5,073,012:
> cross.s <- data.frame(X=seq(demgrid@bbox[1,1]+gridcell/2,
+      demgrid@bbox[1,2]-gridcell/2, gridcell), Y=rep(5073012, demgrid@grid@cells.dim[1]))
> coordinates(cross.s) <- ~ X+Y
# proj4string(cross.s) <- elevations@proj4string
> cross.ov <- overlay(DEM.sim, cross.s)
> plot(cross.ov@coords[,1], cross.ov@data[[1]], type="l", xlab="X",
+      ylab="Z", col="grey")
> for(i in 2:N.sim-1){
>   lines(cross.ov@coords[,1], cross.ov@data[[i]], col="grey")
> }
> lines(cross.ov@coords[,1], cross.ov@data[[N.sim]], lwd=2)
```

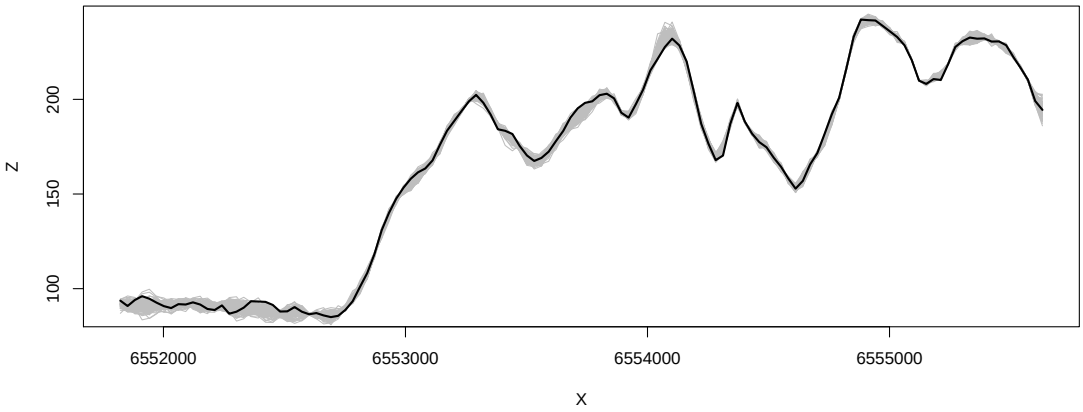

Fig. 10.5: 100 simulations of DEM showing using a cross-section from West to East (cross-section at X=5,073,012). Compare with Temme et al. (2008, p.130).

You will notice that the confidence band is relatively wide (few meters). The confidence band is controlled with the density of sampling locations — the further you get from sampling locations, the higher will be the error. This property we will further explore in §10.5.

Why is a high kappa parameter necessary? If you run DEM simulations with e.g. an exponential model, you will see that the realizations will be much noisier than what we would expect (Hengl et al., 2008). This will happen even if you set the nugget parameters at zero (smooth feature). There are several explanations for this. Having a non-zero grid resolution implies that the correlation between adjacent grid cells is not equal to 1, so that grids may still appear to have noise (Temme et al., 2008). A noisy DEM leads to completely different drainage networks — the streams will be shorter and more random — which we know does not fit knowledge about the area. The Matérn variogram model (Eq.1.3.10), on the other hand, allow us to produce smoother DEMs, while still using objectively estimated nugget, sill and range parameters. This makes it especially suitable for modeling of land surface.

10.4 Generation of stream networks

Now that we have simulated N DEMs, we can derive stream networks using the "*Channel Network*" function, which is available also via the command line i.e. via the `ta_channels` SAGA library. To get complete info about this module you can use:

```
> rsaga.get.usage("ta_channels", 0)
```

First, convert the maps to SAGA format:

```
# write simulated DEMs to SAGA format:
> for(i in 1:N.sim){
>   write.asciigrid(DEM.sim[i], paste("DEM", i, ".asc", sep=""), na.value=-1)
> }
# get a list of files:
> dem.list <- list.files(getwd(), pattern="DEM[[:digit:]]*.asc")
> rsaga.esri.to.sgrd(in.grids=dem.list, out.sgrd=set.file.extension(dem.list,
+     ".sgrd"), in.path=getwd(), show.output.on.console=FALSE)
> unlink(dem.list)
```

Recall that, before we derive a stream network, we also want to remove[7] spurious sinks. For this we can use e.g. the method of Planchon and Darboux (2001):

[6] See Diggle and Ribeiro Jr (2007, p.51–53).

[7] This is also based on the empirical knowledge: water typically erodes small obstacles and creates continuous paths.

```
> stream.list <- list(rep(NA, N.sim))
> for (i in 1:N.sim) {
# First, filter the spurious sinks:
>  rsaga.geoprocessor(lib="ta_preprocessor", module=2,
+      param=list(DEM=paste("DEM", i, ".sgrd", sep=""),
+      RESULT="DEMflt.sgrd", MINSLOPE=0.05), show.output.on.console=FALSE)
# Second, extract the channel network:
>  rsaga.geoprocessor(lib="ta_channels", module=0, param=list(ELEVATION="DEMflt.sgrd",
+      CHNLNTWRK=paste("channels", i, ".sgrd", sep=""), CHNLROUTE="channel_route.sgrd",
+      SHAPES="channels.shp", INIT_GRID="DEMflt.sgrd", DIV_CELLS=3, MINLEN=40),
+      show.output.on.console=FALSE)
# read vector maps into R:
>  stream.list[[i]] <- readOGR("channels.shp", "channels")
>  proj4string(stream.list[[i]]) <- elevations@proj4string
> }
```

Here we use arbitrary input parameters for the minimum length of streams (40) and initial grid, but this is not relevant for this exercise. Note that we do not really need all maps, but only the vector map showing the position of streams. Therefore, we can recycle temporary maps in each loop.

Once the processing is finished, we can visualize all derived streams on top of each other:

```
# plot all derived streams on top of each other:
> stream.plot <- as.list(rep(NA, N.sim))
> for(i in 1:N.sim){
>   stream.plot[[i]] <- list("sp.lines", stream.list[[i]])
> }
> lines.plt <- spplot(DEM.sim[1], col.regions=grey(seq(0.5,1,0.025)),
+      scales=list(draw=T), sp.layout=stream.plot, main="100 streams")
```

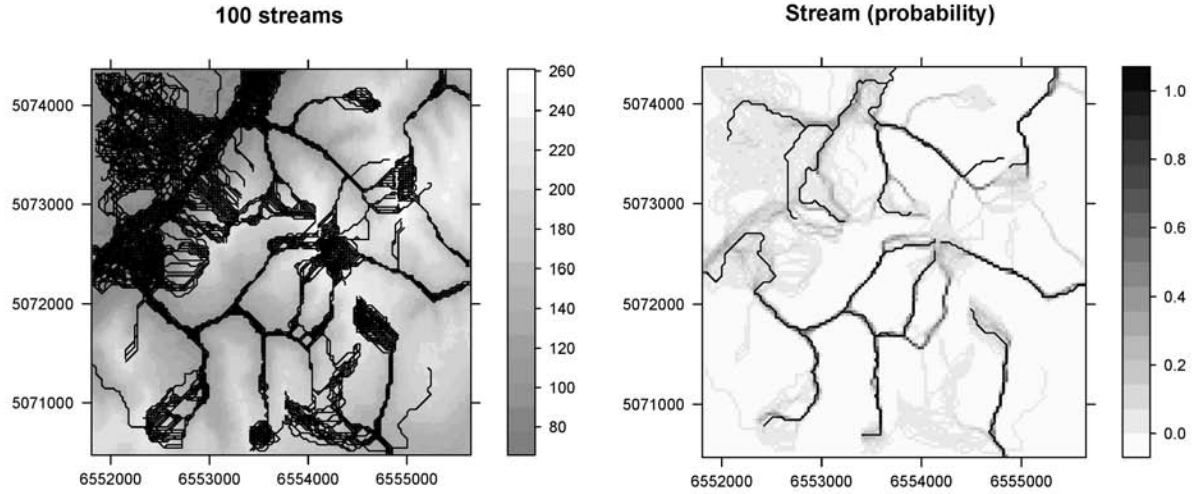

Fig. 10.6: 100 realizations of stream network overlaid on top of each other (left); probability of stream network, overlaid with one realization (right).

which is shown in Fig. 10.6 (left). This visualization of density of streams illustrates the concept of propagated uncertainty. If you zoom in into this map, you will notice that the streams follow the gridded-structure of the DEMs, which explains some artificial breaks in the lines.

To actually derive a probability of mapping a stream, we need to import all gridded maps of streams, then count how many times the model estimated a stream over a certain grid node:

```
# get the list of maps:
> streamgrid.list <- list.files(getwd(), pattern="channels[[:digit:]]*.sgrd")
```

```
> rsaga.sgrd.to.esri(in.sgrds=streamgrid.list,
+    out.grids=set.file.extension(streamgrid.list, ".asc"),
+    out.path=getwd(), prec=0, show.output.on.console=FALSE)
# read all grids into R:
> streamgrid <- readGDAL(set.file.extension(streamgrid.list[[1]], ".asc"))
> streamgrid@data[[1]] <- ifelse(streamgrid$band1<0, 0, 1)
> for(i in 2:length(streamgrid.list)){
>   tmp <- readGDAL(set.file.extension(streamgrid.list[[i]], ".asc"))
# convert to a binary map:
>   streamgrid@data[[i]] <- ifelse(tmp$band1<0, 0, 1)
> }
> names(streamgrid) <- set.file.extension(streamgrid.list, ".asc")
> proj4string(streamgrid) <- elevations@proj4string
```

Now we have a pack of grids (`streamgrid`), with 0/1 values depending on the stream occurrence (yes/no). These can be summed using the `rowSums` method:

```
> streamgrid$pr <- rowSums(streamgrid@data, na.rm=TRUE, dims=1)/length(streamgrid@data)
```

and the probability of detecting a stream is simply the average value of stream from multiple simulations. This map is shown in Fig. 10.6 (right):

```
> stream.plt <- spplot(streamgrid["pr"], col.regions=grey(rev((1:59)/60)),
+      scales=list(draw=T), sp.layout=list("sp.lines", stream.list[[1]]),
+      main="Stream (probability)")
> print(lines.plt, split=c(1,1,2,1), more=TRUE)
> print(stream.plt, split=c(2,1,2,1), more=FALSE)
```

Next, we would like to derive the propagated uncertainty of mapping a stream. Theoretically speaking, stream is a discrete feature that follows a Bernoulli distribution which takes value 1 with success probability p and value 0 with failure probability $q{=}1-p$. Thus the error of mapping a stream can be derived as: $-q\ln(q)-p\ln(p)$, or in R syntax:

```
> streamgrid$pr.var <- -streamgrid$pr*log2(streamgrid$pr)
+               -(1-streamgrid$pr)*log2(1-streamgrid$pr)
```

which means that the highest uncertainty of mapping a stream is when p approaches 0.5 (equal probability of stream and not-stream). As anticipated, the propagated variability of detecting a stream is much higher (in cumulative terms) in the terrace region of the study area (Fig. 10.5). The remaining issue is whether we could explain this variability using some topographic, land surface parameters.

10.5 Evaluation of the propagated uncertainty

Now that we have estimated the propagated uncertainty of extracting channel networks (streams) from DEMs, we can try to understand how this uncertainty relates to the geomorphology of terrain. We will derive and run a comparison by using only a few land surface parameters; you might consider extending the list. First, we can derive mean value (the '*most probable*' DEM) and standard deviation i.e. the propagated uncertainty of mapping elevation:

```
> rsaga.geoprocessor(lib="geostatistics_grid", module=5,
+     param=list(GRIDS=paste(set.file.extension(dem.list,".sgrd"), collapse=";"),
+     MEAN="DEM_avg.sgrd", STDDEV="DEM_std.sgrd"), show.output.on.console=FALSE)
> rsaga.sgrd.to.esri(in.sgrds=c("DEM_avg.sgrd","DEM_std.sgrd"),
+     out.grids=c("DEM_avg.asc","DEM_std.asc"), out.path=getwd(), prec=2)
```

Next, it is interesting to derive a map of slope, as it largely controls the hydrological properties, and the difference from the mean value in 5×5 search radius [8]:

[8] Note that these are wrapper function, which means that they combine several operations together — in this case derivation of slope and conversion of maps to ArcInfo ASCII format.

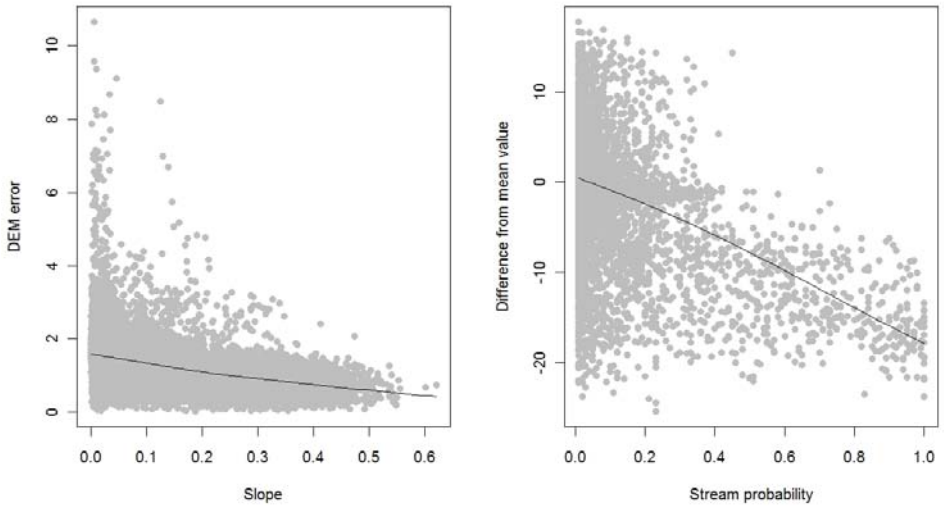

Fig. 10.7: Relationship between the standard error of interpolating elevation and local slope; and probability of deriving streams and difference from the mean elevation.

```
# slope:
> rsaga.esri.wrapper(rsaga.slope, method="poly2zevenbergen", in.dem="DEM_avg.sgrd",
+      out.slope="SLOPE.sgrd", prec=3, clean.up=F)
# residual analysis:
> rsaga.geoprocessor(lib="geostatistics_grid", 0, param=list(INPUT="DEM_avg.sgrd",
+      MEAN="tmp.sgrd", STDDEV="tmp.sgrd", RANGE="tmp.sgrd", DEVMEAN="tmp.sgrd",
+      PERCENTILE="tmp.sgrd", RADIUS=5, DIFF="DIFMEAN.sgrd"))
```

and then read back the results to R:

```
> rsaga.sgrd.to.esri(in.sgrds=c("DEM_avg.sgrd","DEM_std.sgrd", "DIFMEAN.sgrd"),
+    out.grids=c("DEM_avg.asc","DEM_std.asc", "DIFMEAN.asc"), out.path=getwd(), prec=2)
# read back into R:
> gridmaps <- readGDAL("DEM_avg.asc")
> names(gridmaps) <- "DEM"
> gridmaps$std <- readGDAL("DEM_std.asc")$band1
> gridmaps$SLOPE <- readGDAL("SLOPE.asc")$band1
> gridmaps$DIFMEAN <- readGDAL("DIFMEAN.asc")$band1
```

Which allows us to plot the two DEM parameters versus the probability of finding stream and propagated DEM error:

```
> par(mfrow=c(1,2))
> scatter.smooth(gridmaps$SLOPE, gridmaps$std, span=18/19, col="grey",
+     xlab="Slope", ylab="DEM error", pch=19)
> scatter.smooth(streamgrid$pr[streamgrid$pr>0], gridmaps$DIFMEAN[streamgrid$pr>0],
+     span=18/19, col="grey", xlab="Stream probability", ylab="Dif. from mean", pch=19)
> dev.off()
```

Fig. 10.7 shows two interesting things: (1) the errors in elevations are largely controlled by the slope; (2) streams are especially difficult to map in areas where the difference from the mean value is high, i.e. close to zero or positive (meaning areas with low local relief or close to concave shapes). This largely reflects our expectation, but it is rewarding to be able to prove these assumptions using hard data.

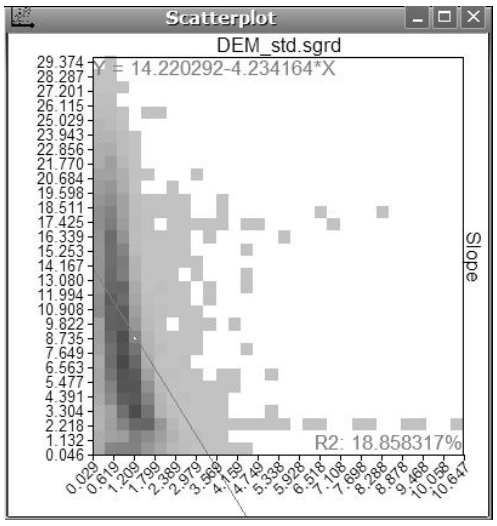

Fig. 10.8: Scatter plot in SAGA GIS. This is the same plot as shown in Fig. 10.7.

Optional: open all derived maps in SAGA and visualize correlations between the probability of streams and DEM error versus various DEM parameters using the scatter plot option. To produce a plot shown in Fig. 10.8 right click on the raster map of interest (`DEM_std`) and then select "*Show scatterplot*" ↦ select grid. You can adjust the size of grid cells and default representation in this plot by editing properties (from the main menu).

10.6 Advanced exercises

10.6.1 Objective selection of the grid cell size

In the previous exercise we set the grid cell size at 30 m, without any real justification. Now we can consider statistically sound approach to select a grid cell size based on the accuracy of the derived stream network. This follows the idea of Hutchinson (1996), who use an iterative DEM cell-size optimization algorithm as implemented in the ANUDEM package. By plotting the error of mapping streams versus the grid spacing index, one can select the grid cell size that shows the maximum information content in the final map. The optimal grid cell size is the one where further refinement does not change the accuracy of derived streams.

We can implement this principle using our case study, i.e. we can derive drainage networks using several grid cell sizes and then see if the spatial location of streams differs significantly from the one derived using the finest resolution (see Fig. 10.9). We can start with generating DEMs from point data (e.g. using splines) using a range of grid cell sizes:

```
> pixel.range <- c(20, 30, 40, 50, 60, 80, 100)
# generate DEMs using splines:
> for(i in 1:length(pixel.range)){
>   rsaga.geoprocessor(lib="grid_spline", module=1,
+      param=list(GRID=paste("DEMpix", pixel.range[i], ".sgrd", sep=""),
+      SHAPES="elevations.shp", FIELD=1, RADIUS=sqrt(areaSpatialGrid(demgrid))/3,
+      SELECT=1, MAXPOINTS=10, TARGET=0, USER_CELL_SIZE=pixel.range[i],
+      USER_X_EXTENT_MIN=demgrid@bbox[1,1]+pixel.range[i]/2,
+      USER_X_EXTENT_MAX=demgrid@bbox[1,2]-pixel.range[i]/2,
+      USER_Y_EXTENT_MIN=demgrid@bbox[2,1]+pixel.range[i]/2,
+      USER_Y_EXTENT_MAX=demgrid@bbox[2,2]-pixel.range[i]/2))
> }
```

Next, we can derive stream networks for each DEM, and then buffer distance to the stream network using a loop. We set the minimum length of stream (`min.len`) based on the cell size of DEM:

```
# estimate drainage map for each DEM:
> for(i in 1:length(pixel.range)){
# filter the spurious sinks:
>   rsaga.geoprocessor(lib="ta_preprocessor", module=2,
+      param=list(DEM=paste("DEMpix", pixel.range[i], ".sgrd", sep=""),
+      RESULT="DEMflt.sgrd", MINSLOPE=0.05), show.output.on.console=FALSE)
# minimum length:
>   min.len <- round(sqrt(areaSpatialGrid(demgrid))/(pixel.range[i]*3.5), 0)
# extract the channel network:
>   rsaga.geoprocessor(lib="ta_channels", module=0,
+      param=list(ELEVATION="DEMflt.sgrd", CHNLNTWRK=paste("chnlntwrk_pix",
+      pixel.range[i], ".sgrd", sep=""), CHNLROUTE="tmp.sgrd",
+      SHAPES=paste("channels_pix", i, ".shp", sep=""), INIT_GRID="DEMflt.sgrd",
```

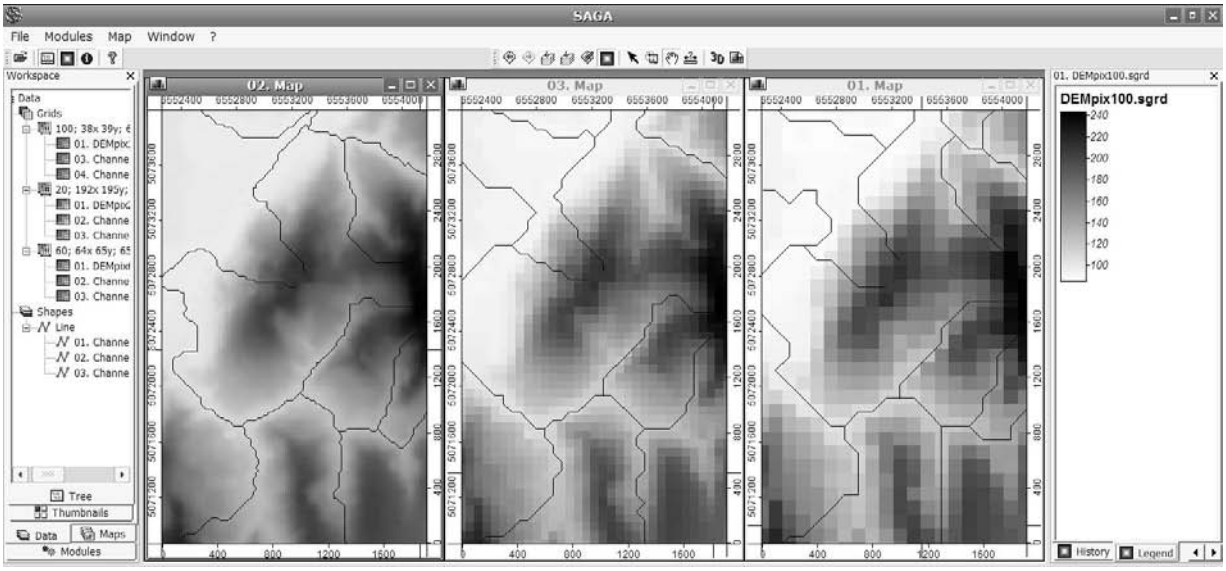

Fig. 10.9: Drainage network derived using different grid cell sizes.

```
+      DIV_CELLS=3, MINLEN=min.len), show.output.on.console=FALSE)
# buffer distance to actual streams (use the finest grid cell size):
>   rsaga.geoprocessor(lib="grid_gridding", module=0,
+      param=list(GRID="stream_pix.sgrd",
+      INPUT=paste("channels_pix", i, ".shp", sep=""), FIELD=1, LINE_TYPE=0,
+      USER_CELL_SIZE=pixel.range[1],
+      USER_X_EXTENT_MIN=demgrid@bbox[1,1]+pixel.range[1]/2,
+      USER_X_EXTENT_MAX=demgrid@bbox[1,2]-pixel.range[1]/2,
+      USER_Y_EXTENT_MIN=demgrid@bbox[2,1]+pixel.range[1]/2,
+      USER_Y_EXTENT_MAX=demgrid@bbox[2,2]-pixel.range[1]/2),
+      show.output.on.console=FALSE)
# extract a buffer distance map:
>   rsaga.geoprocessor(lib="grid_tools", module=10,
+      param=list(SOURCE="stream_pix.sgrd", DISTANCE="tmp.sgrd",
+      ALLOC="tmp.sgrd", BUFFER=paste("buffer_pix", pixel.range[i], ".sgrd", sep=""),
+      DIST=2000, IVAL=pixel.range[1]), show.output.on.console=FALSE)
> }
```

1 and then read maps into R:

```
> rsaga.sgrd.to.esri(in.sgrds="chnlntwrk_pix20.sgrd",
+     out.grids="chnlntwrk_pix20.asc", out.path=getwd(), prec=1)
> griddrain <- readGDAL("chnlntwrk_pix20.asc")
> names(griddrain) <- "chnlntwrk"
> for(i in 2:length(pixel.range)){
>   rsaga.sgrd.to.esri(in.sgrds=paste("buffer_pix", pixel.range[i], ".sgrd", sep=""),
+       out.grids=paste("buffer_pix", pixel.range[i], ".asc", sep=""),
+       out.path=getwd(), prec=1)
>   griddrain@data[paste("buffer_pix", pixel.range[i], sep="")] <-
+       readGDAL(paste("buffer_pix", pixel.range[i], ".asc", sep=""))$band1
> }
> str(griddrain@data)

'data.frame':   37440 obs. of  7 variables:
 $ chnlntwrk    : num  NA NA NA NA NA NA NA NA NA NA ...
 $ buffer_pix30 : num  500 500 480 460 460 440 420 420 400 400 ...
 $ buffer_pix40 : num  500 480 460 460 440 420 420 400 400 380 ...
 $ buffer_pix50 : num  520 500 500 480 460 460 440 440 420 420 ...
```

```
$ buffer_pix60 : num  540 540 520 500 500 480 460 440 440 420 ...
$ buffer_pix80 : num  500 480 460 440 440 420 400 400 380 380 ...
$ buffer_pix100: num  500 480 460 460 440 420 420 4
```

which shows distance from the channel network derived using the finest resolution and increasingly coarser resolutions (30, 40, ...100 m). This allows us to compare how much the stream networks deviate from the '*true*' stream:

```
# summary statistics:
> stream.dist <- as.list(rep(NA, length(pixel.range)))
> mean.dist <- c(0, rep(NA, length(pixel.range)-1))
> for(i in 2:length(pixel.range)){
>   stream.dist[[i]] <- summary(griddrain@data[!is.na(griddrain$chnlntwrk),i])
>   mean.dist[i] <- stream.dist[[i]][4]
> }
# final plot:
> plot(pixel.range, mean.dist, xlab="Grid cell size", ylab="Error", pch=19)
> lines(pixel.range, mean.dist)
```

which will produce the plot shown in Fig. 10.10. Surprisingly, resolution of 50 m is even better than the 30 m resolution; with coarser resolutions (>50 m) the spatial accuracy of mapping streams progressively decreases (average error already >70 m). Note also that these results indicate that there there are '*jumps*' in the accuracy: the stream extraction algorithm generates similar results for resolutions 30–50 m, then it again stabilizes at 80–100 m resolutions.

It appears that we could save a lot of processing time if we would use a resolution of 50 m (instead of 30 m) for this type of modeling. Note that we estimated the error using a single realization of the DEM. One would again need to run such analysis using simulated DEMs at different resolutions, to prove that this difference in accuracy for different grid cell sizes is not an accident.

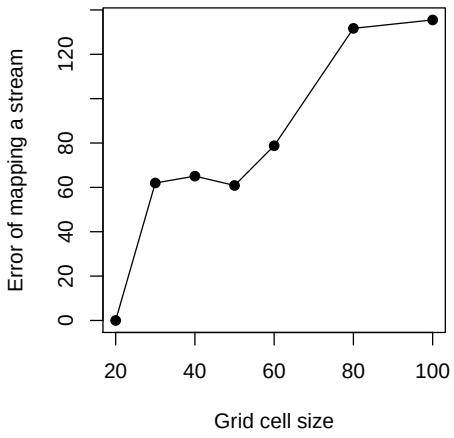

Fig. 10.10: Mean location error of stream network for varying grid cell size (values of both coordinates are in meters).

10.6.2 Stream extraction in GRASS

An equally good alternative to SAGA for processing of elevation data and extraction of DEM parameters and hydrological features is GRASS GIS (Neteler and Mitasova, 2008). We will now run, just for a comparison, extraction of streams in GRASS using the same data set[9]. First, you need to obtain and install GRASS GIS to your computer. After you have finished installing GRASS, switch to your R session and start the spgrass6[10] library that will allow us to control GRASS from R:

```
> library(spgrass6) # version => 0.6-1

  Loading required package: XML
  GRASS GIS interface loaded with GRASS version: (GRASS not running)
```

Because we do not want to worry where GRASS saves temporary files, we can simply assign the environmental parameters to some temporary directory:

```
# Location of your GRASS installation:
> loc <- initGRASS("C:/GRASS", home=tempdir())
> loc
```

[9] The following examples are based on the Windows XP OS. There can be large differences between different OS and different versions of GRASS/spgrass6!

[10] http://cran.r-project.org/web/packages/spgrass6/

```
  gisdbase    c:/WINNT/profiles/software/LOCALS~1/Temp/RtmpdeK5s9
  location    file678418be
  mapset      file3d6c4ae1
  rows        1
  columns     1
  north       1
  south       0
  west        0
  east        1
  nsres       1
  ewres       1
  projection  NA
```

Note that these settings are in fact nonsense. We will soon replace these with the actual parameters once we import an actual raster map. What is important is that we have established a link with GRASS and set the working directory. We can proceed with importing the previously derived DEM into GRASS:

```
> parseGRASS("r.in.gdal") # commmand description

  Command: r.in.gdal
  Description: Import GDAL supported raster file into a binary raster map layer.
  Keywords: raster, import
  Parameters:
    name: input, type: string, required: no, multiple: no
   [Raster file to be imported]
    ...


# Import the ArcInfo ASCII file to GRASS:
> execGRASS("r.in.gdal", flags="o", parameters=list(input="DEM_avg.asc", output="DEM"))

  WARNING: Over-riding projection check

  RINGDA~1 complete. Raster map <DEM> created.
```

We can use the parameters of the imported map to set up the geographic region:

```
> execGRASS("g.region", parameters=list(rast="DEM"))
> gmeta6()

  gisdbase    c:/WINNT/profiles/software/LOCALS~1/Temp/RtmpdeK5s9
  location    file678418be
  mapset      file3d6c4ae1
  rows        130
  columns     128
  north       5074369
  south       5070469
  west        6551807
  east        6555647
  nsres       30
  ewres       30
  projection  NA
```

which is the averaged DEM from multiple realizations derived in §10.5. Note that Windows system generates the temporary name of the mapset and location. Again, we do not worry too much about this because we use GRASS as an external application to run geographical analysis; temporary files will be recycled, at the end we will read only the final results back into R.

We proceed with the extraction of the drainage network:

```
# extract the drainage network:
> execGRASS("r.watershed", flags=c("m", "overwrite"),
+     parameters=list(elevation="DEM", stream="stream", threshold=as.integer(50)))
```

```
 SECTION 1 beginning: Initiating Variables. 5 sections total.
 SECTION 1b (of 5): Determining Offmap Flow.

 SECTION 2: A * Search.

 SECTION 3: Accumulating Surface Flow.

 SECTION 4: Watershed determination.

 SECTION 5: Closing Maps.
```

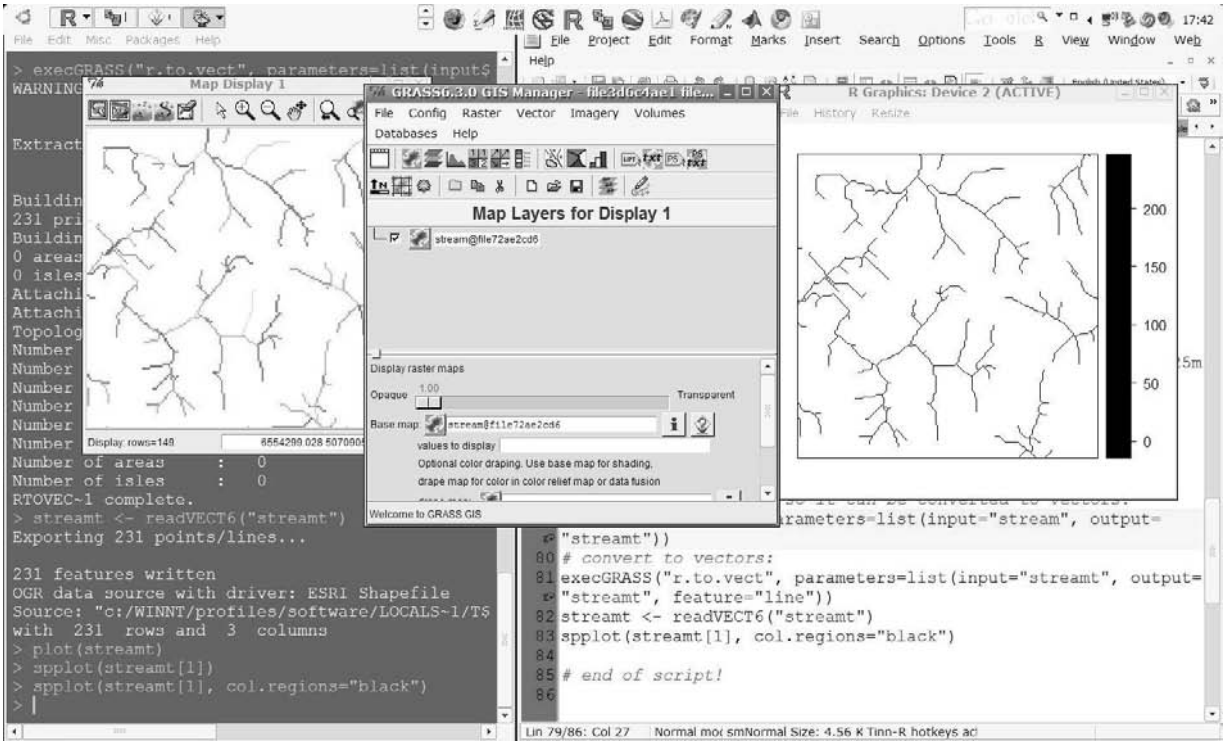

Fig. 10.11: Example of a screen shot — drainage extraction steps using the Baranja hill data set in GRASS GIS.

which will generate a raster map showing the position of streams (Fig. 10.11). Note that GRASS typically generates a rich output with many technical details of interest to a specialist. Before we can convert the derived map to a vector layer, we need to thin it:

```
> execGRASS("r.thin", parameters=list(input="stream", output="streamt"))

 File stream -- 130 rows X 128 columns
 Bounding box: l = 2, r = 129, t = 2, b = 131
 Pass number 1
 Deleted 55 pixels
 Pass number 2
 Deleted 0 pixels
 Thinning completed successfully.
 Output file 130 rows X 128 columns
 Window 130 rows X 128 columns


# convert to vectors:
> execGRASS("r.to.vect", parameters=list(input="streamt",
+            output="streamt", feature="line"))
```

```
WARNING: Default driver / database set to:
         driver: dbf
         database: $GISDBASE/$LOCATION_NAME/$MAPSET/dbf/
Extracting lines...

Building topology for vector map <streamt>...
Registering primitives...

165 primitives registered
652 vertices registered
Building areas...

0 areas built
0 isles built
Attaching islands...
Attaching centroids...

Number of nodes: 176
Number of primitives: 165
Number of points: 0
Number of lines: 165
Number of boundaries: 0
Number of centroids: 0
Number of areas: 0
Number of isles: 0
RTOVEC ~1 complete.
```

Processing is now complete, so we can read the produced map from R. For this, we use the generic spgrass6 command that reads any type of GRASS vector:

```
# read the generated stream network map into R:
> streamt <- readVECT6("streamt")

Exporting 165 points/lines...

165 features written
OGR data source with driver: ESRI Shapefile
Source: "c:/WINNT/profiles/software/LOCALS~1/Temp/RtmpdeK5s9/file3d6c4ae1/.tmp",
layer: "streamt"
with  165  rows and  3  columns
Feature type: wkbLineString with 2 dimensions


> plot(streamt)
```

which shows a similar result as shown in Fig. 10.1. You can now open all maps you have generated and visualize them in GRASS. Fig. 10.11 will give you some idea about the GRASS interface. In summary, there are noticable differences in the ways the things are run with SAGA and GRASS. GRASS seem to be more demanding considering the control of the package. On the other hand, it is a much larger and more international project than SAGA. It is really a question of taste if one prefers to use one or the other, but there are also no obstacles to combine them.

10.6.3 Export of maps to GE

In the final step we will export the stream probability map from R to Google Earth, this time without using SAGA GIS[11]. We can start by re-projecting the derived grid to the latitude longitude system:

```
> streamgrid.ll <- spTransform(streamgrid["pr"], CRS("+proj=longlat +datum=WGS84"))
> streamgrid.ll@bbox
```

[11] This is somewhat more complicated. Compare with §5.6.2.

```
        min      max
x 18.66122 18.71065
y 45.77646 45.81158
```

which will create a point map (not a grid!), which means that we need to create the grid topology in the `longlat` coordinate system ourselves. To do this, we first need to estimate the grid cell size in arcdegrees, e.g. by using the Eq.(3.3.1) explained in §3.3.1. The width correction factor[12] based on the latitude of the center of the study area, can be estimated as:

```
> corrf <- (1 + cos((streamgrid.ll@bbox[1, "max"] +
+          streamgrid.ll@bbox[2, "min"])/2 * pi/180))/2
```

and then the grid cell size in arcdegrees is approximately:

```
> geogrd.cell <- corrf*(streamgrid.ll@bbox[1,"max"] -
+          streamgrid.ll@bbox[1, "min"]) / streamgrid@grid@cells.dim[1]
> geogrd.cell
```

```
  [1] 0.0003564123
```

which means that a width of a 30 m pixel at this latitude corresponds to about 1.3 arcseconds.

Fig. 10.12: Baranja hill and derived stream probability, visualized in Google Earth.

Once we have estimated the grid cell size in geographical coordinates, we can also generate the new grid system:

```
> geoarc <- spsample(streamgrid.ll, type ="regular", cellsize=c(geogrd.cell, geogrd.cell))
> gridded(geoarc) <- TRUE
> gridparameters(geoarc)
```

```
   cellcentre.offset     cellsize
x1          18.66127 0.0003564123
x2          45.77662 0.0003564123
   cells.dim
x1       139
x2        99
```

[12] For data sets in geographical coordinates, a cell size correction factor can be estimated as a function of the latitude and spacing at the equator.

```
> geoarc.grid <- SpatialGridDataFrame(geoarc@grid,
+       data=data.frame(rep(1, length(geoarc@grid.index))),
+       proj4string=streamgrid.ll@proj4string)
```

Now we need to estimate values of our target variable at the new grid nodes. We use the `interp` method as implemented in the akima package, which leads to a simple bilinear resampling:

```
> library(akima)
> streamgrid.llgrd <- interp(x=streamgrid.ll@coords[,1], y=streamgrid.ll@coords[, 2],
+    z=streamgrid.ll$pr, xo=seq(geoarc.grid@bbox[1, "min"], geoarc.grid@bbox[1, "max"],
+    length=geoarc.grid@grid@cells.dim[[1]]), yo=seq(geoarc.grid@bbox[2, "min"],
+    geoarc.grid@bbox[2, "max"], length=geoarc.grid@grid@cells.dim[[2]]),
+    linear=TRUE, extrap=FALSE)
# convert to sp class:
> streamgrid.llgrd <- as(as.im(streamgrid.llgrd), "SpatialGridDataFrame")
> proj4string(streamgrid.llgrd) <- CRS("+proj=longlat +datum=WGS84")
# mask the "0" values:
> streamgrid.llgrd$pr <- ifelse(streamgrid.llgrd$v < 0.05, NA, streamgrid.llgrd$v)
```

which allows us to finally generate a KML ground overlay for Google Earth (Fig. 10.12):

```
> streamgrid.kml <- GE_SpatialGrid(streamgrid.llgrd)
> png(file="stream.png", width=streamgrid.kml$width,
+     height=streamgrid.kml$height, bg="transparent")
> par(mar=c(0, 0, 0, 0), xaxs="i", yaxs="i")
> image(as.image.SpatialGridDataFrame(streamgrid.llgrd["pr"]),
+     col=rev(), xlim=streamgrid.kml$xlim,
+     ylim=streamgrid.kml$ylim)
> kmlOverlay(streamgrid.kml, "stream.kml", "stream.png",
+     name="Stream probability")

  [1] "<?xml version='1.0' encoding='UTF-8'?>"
  [2] "<kml xmlns='http://earth.google.com/kml/2.0'>"
  [3] "<GroundOverlay>"
  [4] "<name>Stream probability</name>"
  [5] "<Icon><href>stream.png</href><viewBoundScale>0.75</viewBoundScale></Icon>"
  [6] "<LatLonBox><north>45.8119041798664</north><south>45.7762593102449</south>
       <east>18.7108375328584</east><west>18.6609075254189</west></LatLonBox>"
  [7] "</GroundOverlay></kml>"


> dev.off()

 windows
       2
```

Visualization of generated maps in Google Earth is important for several reasons: (1) we can check if the coordinate system definition is correct; (2) we can evaluate and interpret the results of mapping using high resolution satellite imagery; (3) Google Earth allows 3D exploration of data, which is ideal for this type of exercise (read more in §3.3.1).

Before closing the R session, it is also advisable to clean up all the temporary files:

```
> save.image(".RData")
> unlink("*.hgrd")
> unlink("*.sgrd")
> unlink("*.sdat")
> unlink("DEM**.***")
> unlink("channels**.***")
```

Self-study exercises:

(1.) What is the sampling density of the `elevations` map in no./ha? (HINT: divide number of points per size of area.)

(2.) How precise is the interpolation of elevations overall following the ordinary kriging model? (HINT: run ordinary kriging and derive mean value of the kriging variance for the whole area.)

(3.) What is the inter-quantile range of derived stream probability? (HINT: derive summary statistics and then take the first and third quantile.)

(4.) How does the precision of generating DEM changes considering the edge of area? (HINT: plot the standard deviation of generated DEMs versus the edge contamination map that you can derive in SAGA GIS.)

(5.) How does the probability of mapping streams change with PLANC? (HINT: derive a correlation coefficient between propagated error and mapped values; plot a scatter plot.)

(6.) What is the percentage of area with stream probability >0.5?

(7.) How much is 50 m in arcdegrees for this study area? And in arcsecond?

Further reading:

★ Hengl, T., Bajat, B., Reuter, H. I., Blagojević, D., 2008. Geostatistical modelling of topography using auxiliary maps. Computers & Geosciences, 34: 1886–1899.

★ Hengl, T., Reuter, H. (Eds.), 2008. **Geomorphometry: Concepts, Software, Applications**. Vol. 33 of Developments in Soil Science. Elsevier, Amsterdam, p. 772.

★ Heuvelink, G. B. M. 2002. Analysing uncertainty propagation in GIS: why is it not that simple?. In: Foody, G. M., Atkinson, P. M. (Eds.), Uncertainty in Remote Sensing and GIS, Wiley, Chichester, pp. 155–165.

★ Temme, A. J. A. M., Heuvelink, G. B. M., Schoorl, J. M., Claessens, L., 2008. Geostatistical simulation and error propagation in geomorphometry. In: Hengl, T., Reuter, H. I. (Eds.), Geomorphometry: Concepts, Software, Applications, volume 33: Developments in Soil Science. Elsevier, Amsterdam, 121–140.

★ `http://geomorphometry.org` — The Geomorphometry research group.