Priority Flow Workflow Example

This notebook walks through a typical workflow for processing a DEM to ensure a fully connected hydrologic drainage network. For, more details on the Priority Flow tool refer to Condon and Maxwell (2019) (https://doi.org/10.1016/j.cageo.2019.01.020 (https://doi.org/10.1016/j.cageo.2019.01.020))

For this example we will use the sample watershed used in Condon and Maxwell (2019). The DEM and mask files for this domain are provided with the Priority Flow library.

Background Information and Setup

First you will need to load the Priority Flow library and the fields library for plotting. If you haven't installed the PriorityFlow library yet you can learn how in the ReadMe of the GitHub repo

```
library(PriorityFlow)
library(fields)
```

You can learn more about the library and its functions using the help function and clicking on the index:

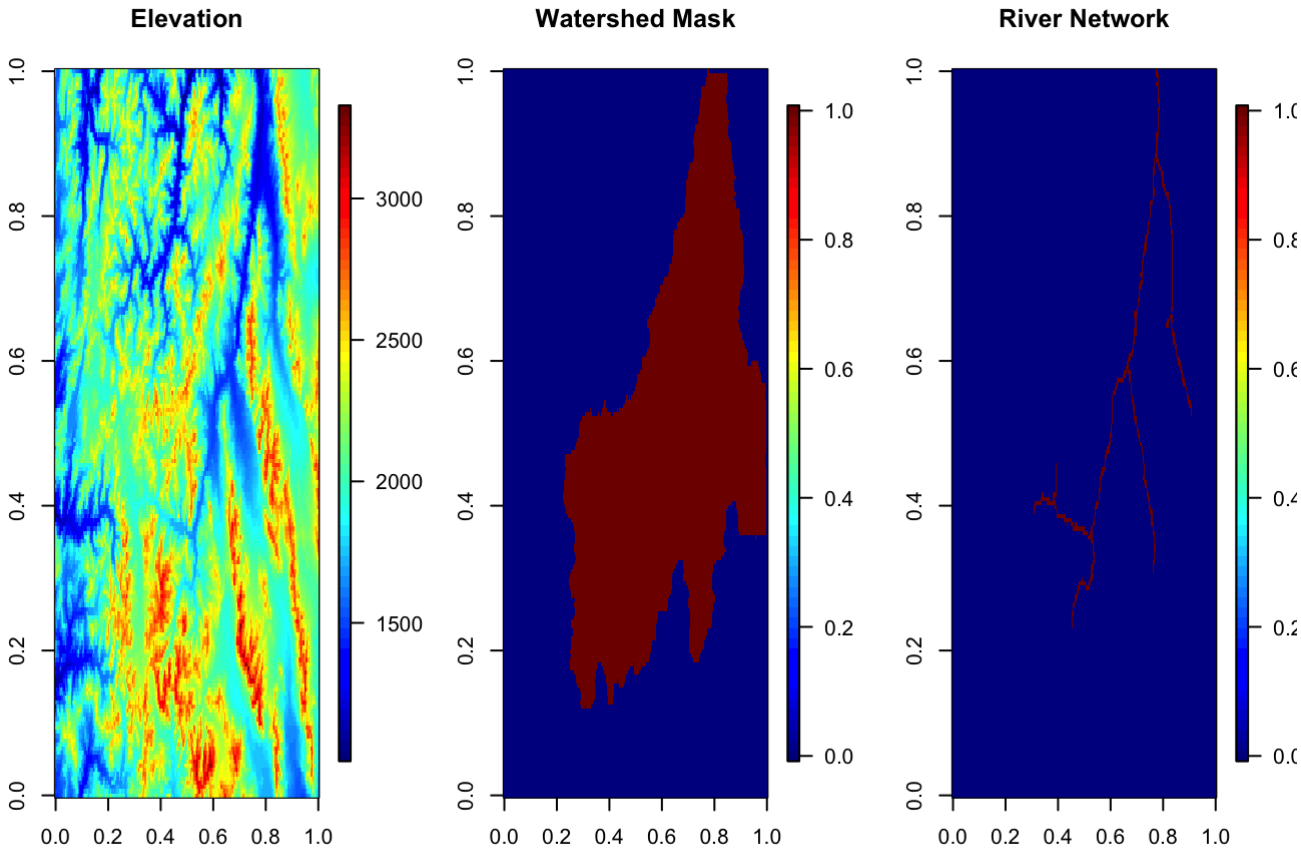
```
help("PriorityFlow")
```

These are the three inputs we will be using for our domain processing:

1. The unprocessed DEM (Digital Elevation Model)
2. A mask of the watershed we are interested in
3. A mask of our desired drainage network

NOTE: the only required input is an DEM the other two are optional depending on how you would like to process things.

```
par(mfrow=c(1,3))
image.plot(DEM, main="Elevation")
image.plot(watershed.mask, main="Watershed Mask")
image.plot(river.mask, main="River Network")
```




```
nx=dim(DEM)[1]
ny=dim(DEM)[2]
```

NOTE: The data sets used here are already formatted as [nx,ny] matrices such that the [i,j] indices correspond to the x and y axes of the domain respectively and [1,1] is the lower left corner and [nx,ny] is the upper right corner. You will need to have your DEM and any other input files formatted like this before you get started. To check that you have them formatted correctly you should be able to do image.plot(yourDEM) and your domain should appear correctly (i.e. untransposed).

Step 1: Processing the DEM

In this step the priority flow algorithm will be used to traverse the raw DEM and ensure that every grid cell in the domain has a pathway to exit along D4 drainage pathways (i.e. there are no unintended internal sinks).

The primary outcomes of this step are:

1. A processed DEM where elevations have been adjusted to ensure drainage
2. A map of flow directions for every grid cell (Unless a different numbering scheme is specified the following will be used: 1=down, 2=left, 3=up, 4=right)

There are several ways to complete this processing. They are listed here in order of increasing complexity. The primary difference between these approaches is whether you will be limiting processing to some watershed mask and how you initialize the processing. By default Priority Flow will find all of the border cells and set these as the targets ensuring that every other cell in the domain is able to drain to one of the target cells. The initial list of

target cells is generated with the init queue function. If you know where the drainage points are in your domain or if you have an internally draining basin where you would like to intentionally set an internal cell as a target point you can make these modifications with the initialization step.

Option 1: If you have rectangular DEM and all border cells will be used as target exit points

In this case PriorityFlow will ensure that every grid cell in the domain drains to the edge of the domain without identifying the desired drainage points a priori

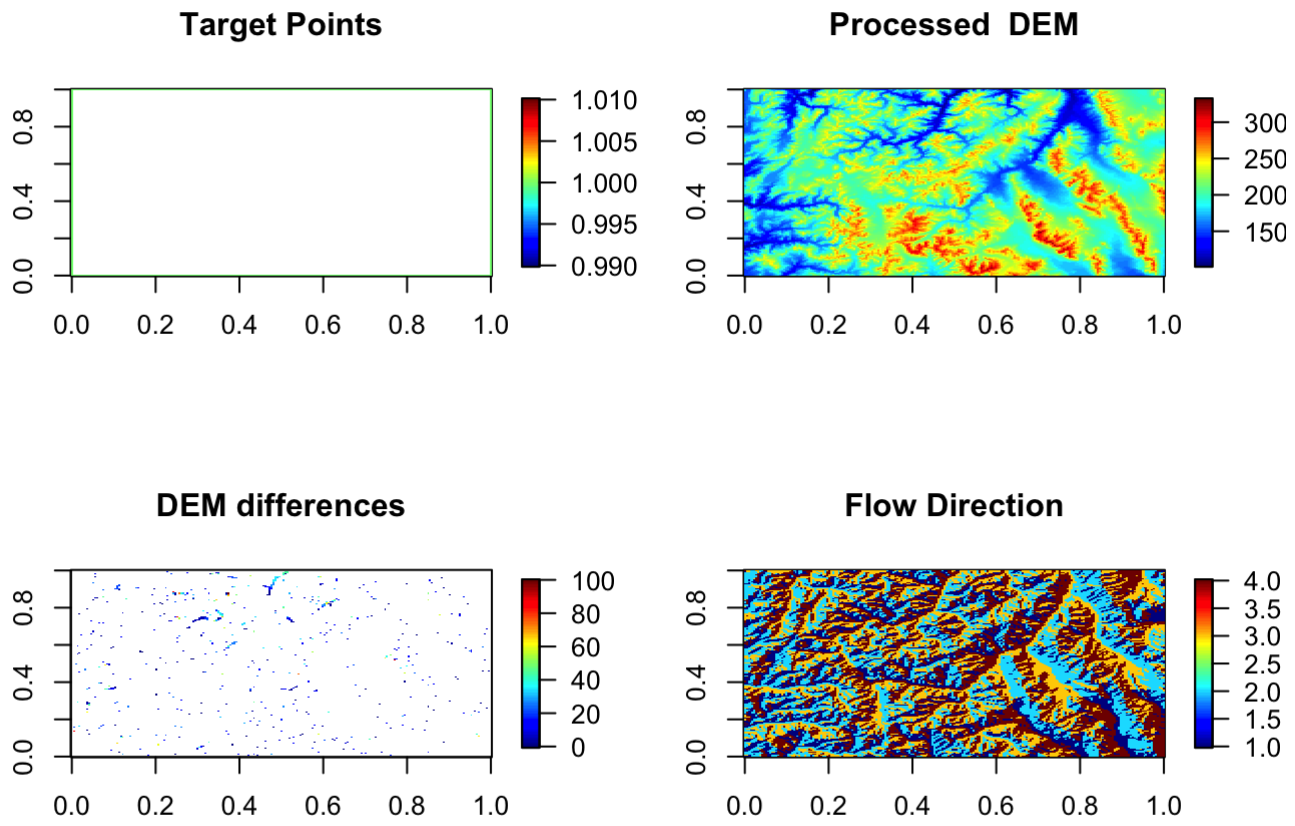
```
#setup target points
init=InitQueue(DEM)
```

```
## [1] "No init mask provided all border cells will be added to queue"
## [1] "No domain mask provided using entire domain"
## [1] "No border provided, setting border using domain mask"
```

```
#process the DEM
travHS=D4TraverseB(DEM, init$queue, init$marked, basins=init$basins,  epsilon=0, nchunk=
10)

#some calculations for plotting
dif=travHS$dem-DEM
dif[dif==0]=NA
targets=init$marked
targets[targets==0]=NA

#plotting
par(mfrow=c(2,2))
image.plot(targets, main="Target Points")
image.plot(travHS$dem, main="Processed  DEM")
image.plot(dif, main="DEM differences")
image.plot(travHS$direction, main="Flow Direction")
```



Option 2: If you only want to process your DEM within a pre-defined watershed mask and all border cells of that mask will be used as target exit points

In this case PriorityFlow will define the domain as the area defined by the mask and ensure that every point in the domain drains to one of the edge points.

*Note: The mask file should have values of 1 for cells inside the domain and 0 everywhere else

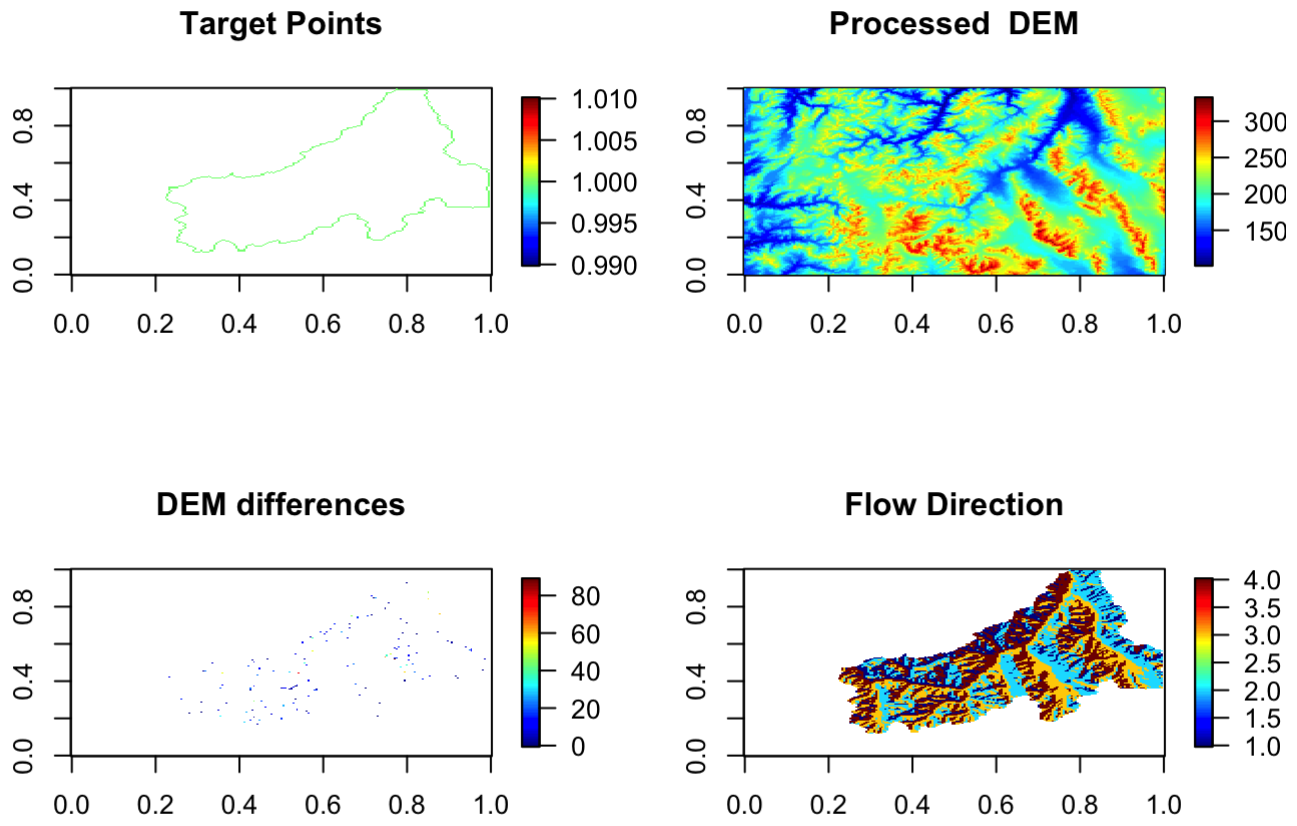
```
#setup target points
init=InitQueue(DEM, domainmask=watershed.mask)
```

```
## [1] "No init mask provided all border cells will be added to queue"
## [1] "No border provided, setting border using domain mask"
```

```
#process the DEM
travHS=D4TraverseB(DEM, mask=watershed.mask, init$queue, init$marked, basins=init$basin
s,  epsilon=0, nchunk=10)

#some calculations for plotting
dif=travHS$dem-DEM
dif[dif==0]=NA
targets=init$marked
targets[targets==0]=NA

#plotting
par(mfrow=c(2,2))
image.plot(targets, main="Target Points")
image.plot(travHS$dem, main="Processed  DEM")
image.plot(dif, main="DEM differences")
image.plot(travHS$direction, main="Flow Direction")
```



Option 3: If you want to have more control over the set of target points used in the processing

Options 1 and 2 both default to using the entire boundary of either the rectangular domain or the watershed mask as the target points for DEM processing. This means that the processing will be counted as complete once every point in the domain has a path to make it to one of the target points. In almost all cases options 1 and 2 will be sufficient. However, you might want more control over the target points in two case:

1. If you have an internally draining basin you might want to specify a point or points on the interior of your domain that count as approved target points. This can be accomplished by providing the InitQueue function with your own border cell file with the border option.
2. If you are using a watershed boundary and you want to be sure that all of the edges of the domain point in except for your outlet point. If this is the case you could provide it with a watershed mask as with option 2 but specify only one border point for the InitQueue with the border option. Or you can provide the InitQueue function with a river mask and a watershed mask and it will identify only those points on the watershed mask that touch a boundary of your domain as target outlet points.

```
#setup target points
init=InitQueue(DEM, domainmask=watershed.mask, initmask=river.mask)
```

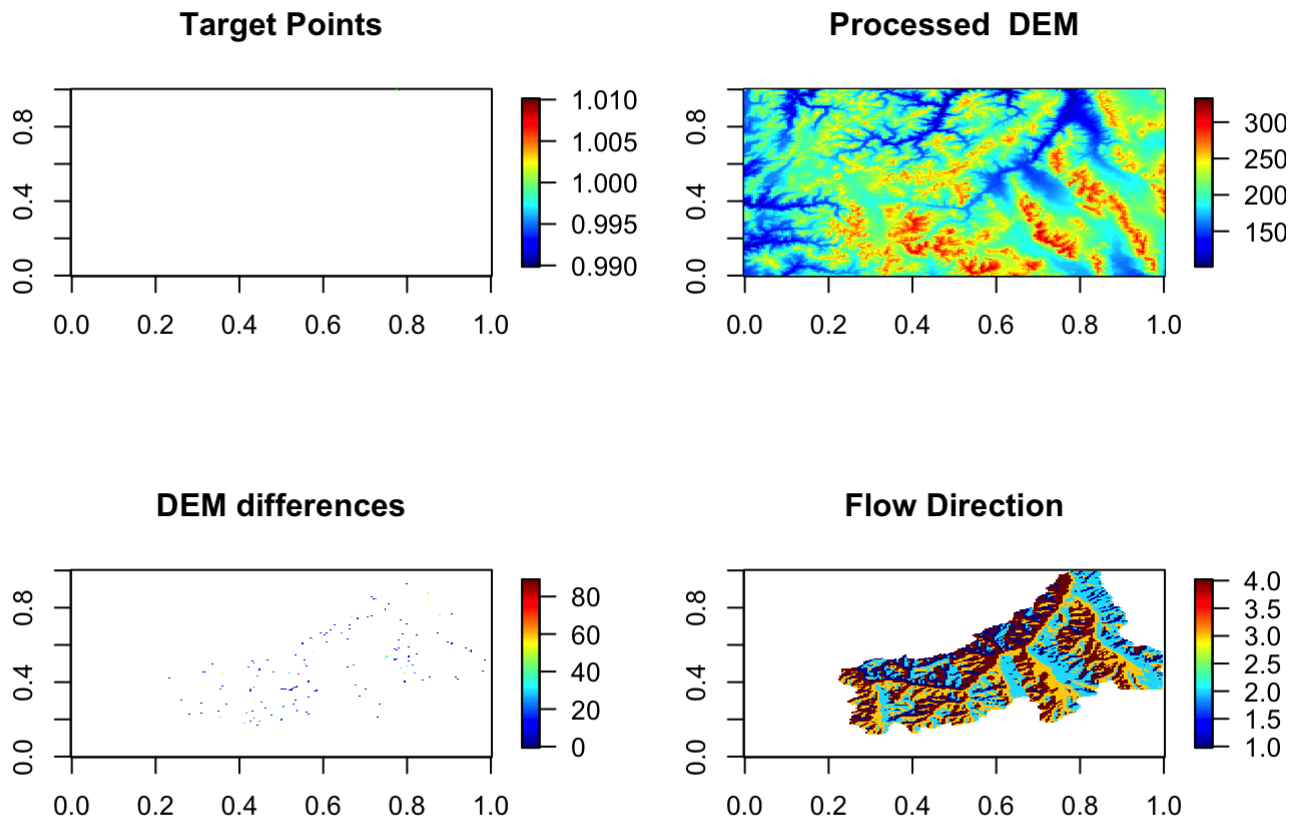
```
## [1] "No border provided, setting border using domain mask"
```

```
#process the DEM
travHS=D4TraverseB(DEM, mask=watershed.mask, init$queue, init$marked, basins=init$basin
s,  epsilon=0, nchunk=10)
```

```
## [1] "inital queue: 3 Not splitting"
```

```
#some calculations for plotting
dif=travHS$dem-DEM
dif[dif==0]=NA
targets=init$marked
targets[targets==0]=NA

#plotting
par(mfrow=c(2,2))
image.plot(targets,  main="Target Points")
image.plot(travHS$dem, main="Processed  DEM")
image.plot(dif, main="DEM differences")
image.plot(travHS$direction, main="Flow Direction")
```



Option 4: If you want to enforce flow paths along a pre-defined drainage network

In most cases the drainage network results from the standard DEM processing illustrated in the options above will be a good match with expected drainage networks. However, if you have a low resolution DEM or are dealing with a very flat domain the noise in the DEM can result in a drainage network that doesn't match with the know drainage network. In this case you can run a modified workflow where you apply priority flow network processing first along a pre-defined drainage network. Essentially processing just the cells on this drainage network to ensure that they drain out of the domain. Then once this is done do a second pass of processing to ensure that every grid cell not on the pre-defined river network can drain to that network. These steps for this workflow are described in detail in Condon and Maxwell (2019) and are documented in Downwinding workflows 3 and 4.

Step 2: Smoothing along the drainage network

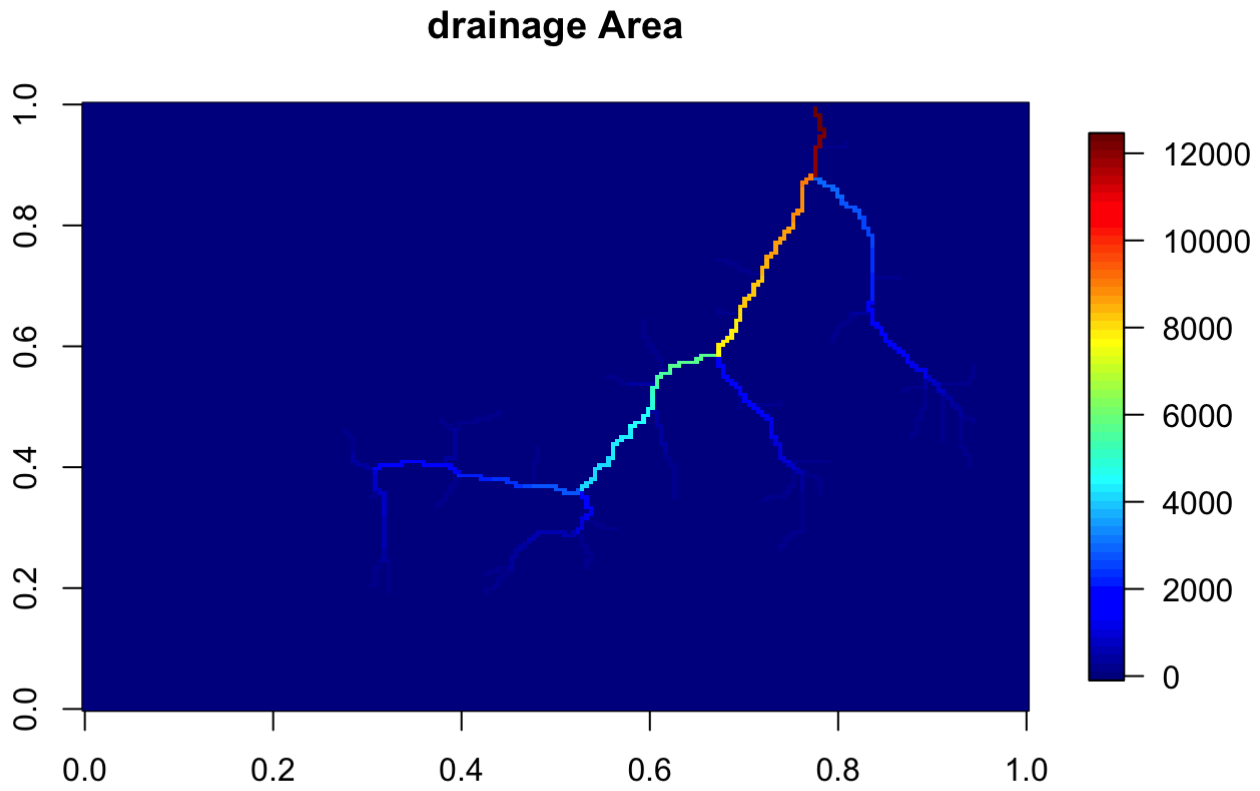
Step 1 will ensure a drainage network which is fully connected and where all grid cells are guaranteed to drain to one of the target points (usually the edge of the domain). However, there can still be significant noise along the drainage path where jumps in the DEM can impact simulated flow performance. To address this you can do some additional smoothing along the river network.

Step 2.1: Calculate drainage areas

Before you can smooth your river network you need to know where it is. To do this you can use the flow direction file created in *Step 1* to calculate the drainage area for every grid cell in the domain

```
area=drainageArea(travHS$direction, mask=watershed.mask, printflag=F)

image.plot(area,main="drainage Area")
```



Note: In this example I used the watershed mask for the area calculations to be consistent with last DEM processing choices made. However, a mask is not required at this step if you did the DEM processing without applying a mask (i.e. as long as you have flow directions for the whole domain)

Step 2.2: Use a drainage area threshold to define a river network

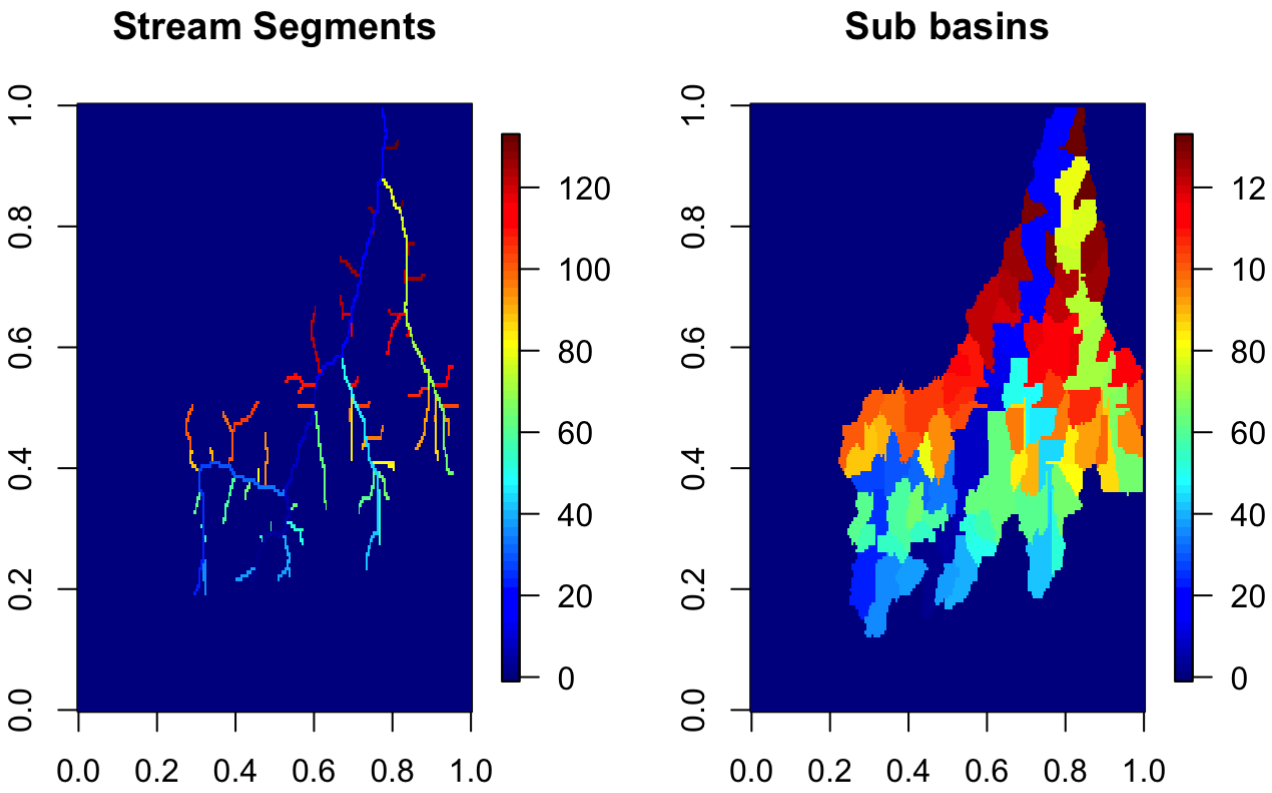
Next you can use the flow direction file and the drainage areas to define a river network given a user specified drainage area threshold to define rivers (i.e. any cell with >=riv_th cells draining to it will count as a river)

```
subbasin=CalcSubbasins(travHS$direction, area=area, mask=watershed.mask, riv_th=60, merg
e_th=10)
```

Note:Try changing the riv_th to change the density of the river network that is created. The merge threshold is used to lump together river segments with drainage areas below this threshold. Use help(CalcSubbasins) to learn more about the options for this function

Take a look at the resulting river network

```
par(mfrow=c(1,2))
image.plot(subbasin$segments, main="Stream Segments")
image.plot(subbasin$subbasins, main="Sub basins")
```



Step 2.3: Smooth the DEM along river segments

Now you can apply smoothing along your river segments This function requires:

1. The processed DEM from the PriorityFlow processing in Step 1
2. The flow directions from the PriorityFlow processing in Step 1
3. Information on the starting and ending points of every river segment calculated with the subbasin function (*subbasin$summary*)
4. A map of the river segments from the subbasin function (*subbasin$segments*)
5. A minimum elevation difference between designated stream cells and non stream cells that are draining to the stream segments (specified here as epsilon).

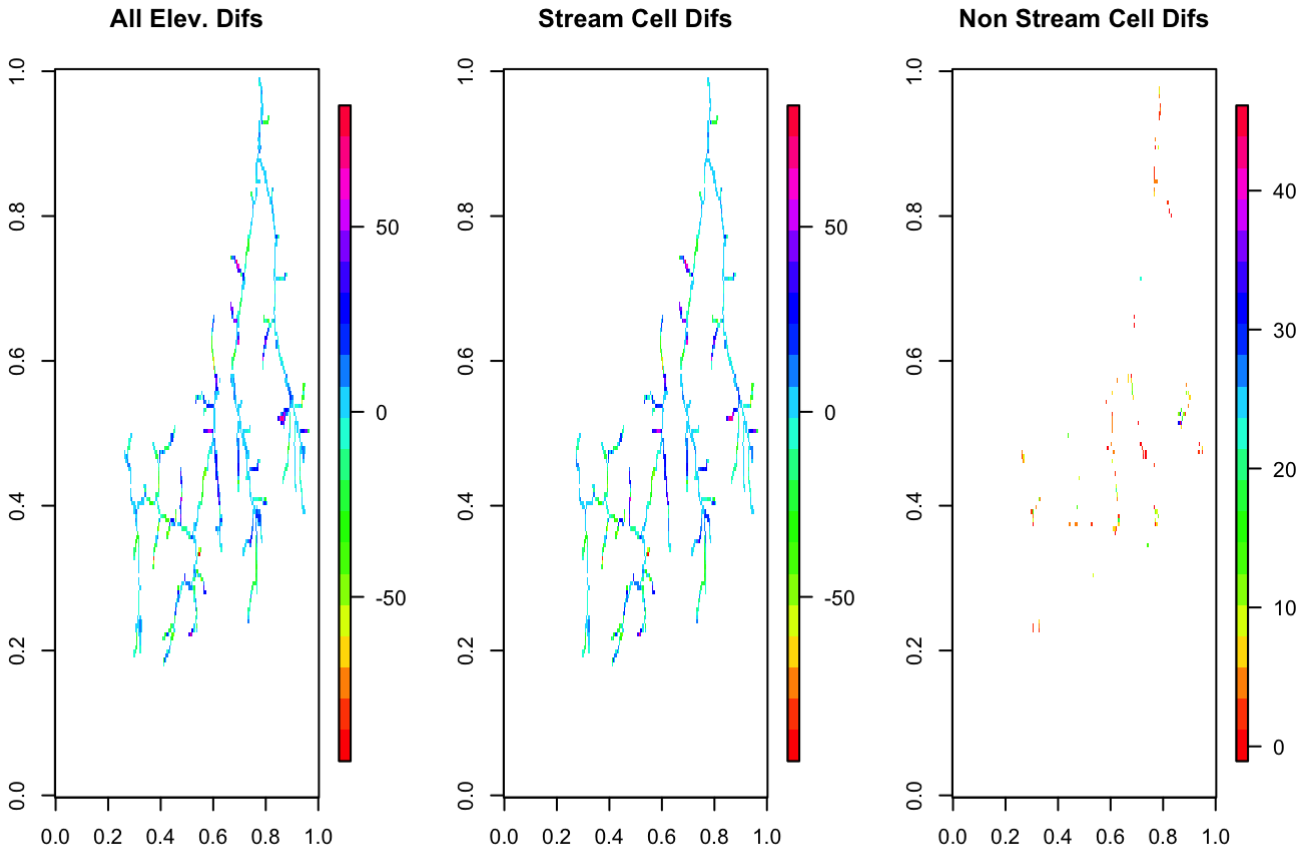
This function will calculate the elevation difference from the start to the end of every segment and apply a constant delta along the length of the segment to get from the top to the bottom. Then it will traverse back up the drainage network checking that in the stream adjustment process no stream cells were raised above their surrounding bank cells using the user specified epsilon value to enforce a difference of at least epsilon between bank and stream cells. IF a bank cell needs to be raised to meet the epsilon threshold the algorithm will continue traversing up the hill slope according to the flow direction file raising cells until every neighboring cell can drain with the required epsilon criteria.

```
RivSmooth=RiverSmooth(dem=travHS$dem, travHS$direction, mask=watershed.mask, river.summa
ry=subbasin$summary, river.segments=subbasin$segments, epsilon=5)
```

Note: The mask is optional for this function too. A watershed mask only needs to be provided here if the flow directions were only calculated within the mask in Step 1.

Now you can do some plotting to see how this smoothing changed the DEM

```
#plot the differences in the DEM along the hillslopes and the rivers
par(mfrow=c(1,3))
  #calcualte elevation differences
  dif=(RivSmooth$dem.adj-travHS$dem)
  #mask out the river mask to plot differences along streams and on hillslopes separatel
y
  rivmask=subbasin$segments
  rivmask[which(rivmask>0)]=1
  hillmask=matrix(1, nrow=dim(rivmask)[1], ncol=dim(rivmask)[2])
  hillmask[which(rivmask>0)]=0
  difhill=dif*hillmask
  difriv=dif*rivmask
  #replacing 0s with NAs
  dif[dif==0]=NA
  difriv[difriv==0]=NA
  difhill[difhill==0]=NA
  #plot
  image.plot(dif, main="All Elev. Difs", col=rainbow(21)) #col=viridis(21))
  image.plot(difriv, main="Stream Cell Difs", col=rainbow(21)) #col=viridis(21))
  #image(rivmask, breaks=c(0,0.5,1.5), col=rev(gray.colors(2)), add=F)
  if(length(which(is.na(difhill)==F))>0){
    image.plot(difhill, main="Non Stream Cell Difs", col=rainbow(21)) #col=viridis(21))
  }
```



Also you can use the *PathExtract* function to walk downstream from any point in the domain and look at how this processing changed elevations along any flow path.

For this example I'm using the subbasin$summary table to pick one of the stream segment starts as my starting point but you could give the PathExtract function any point in the domain as its starting point

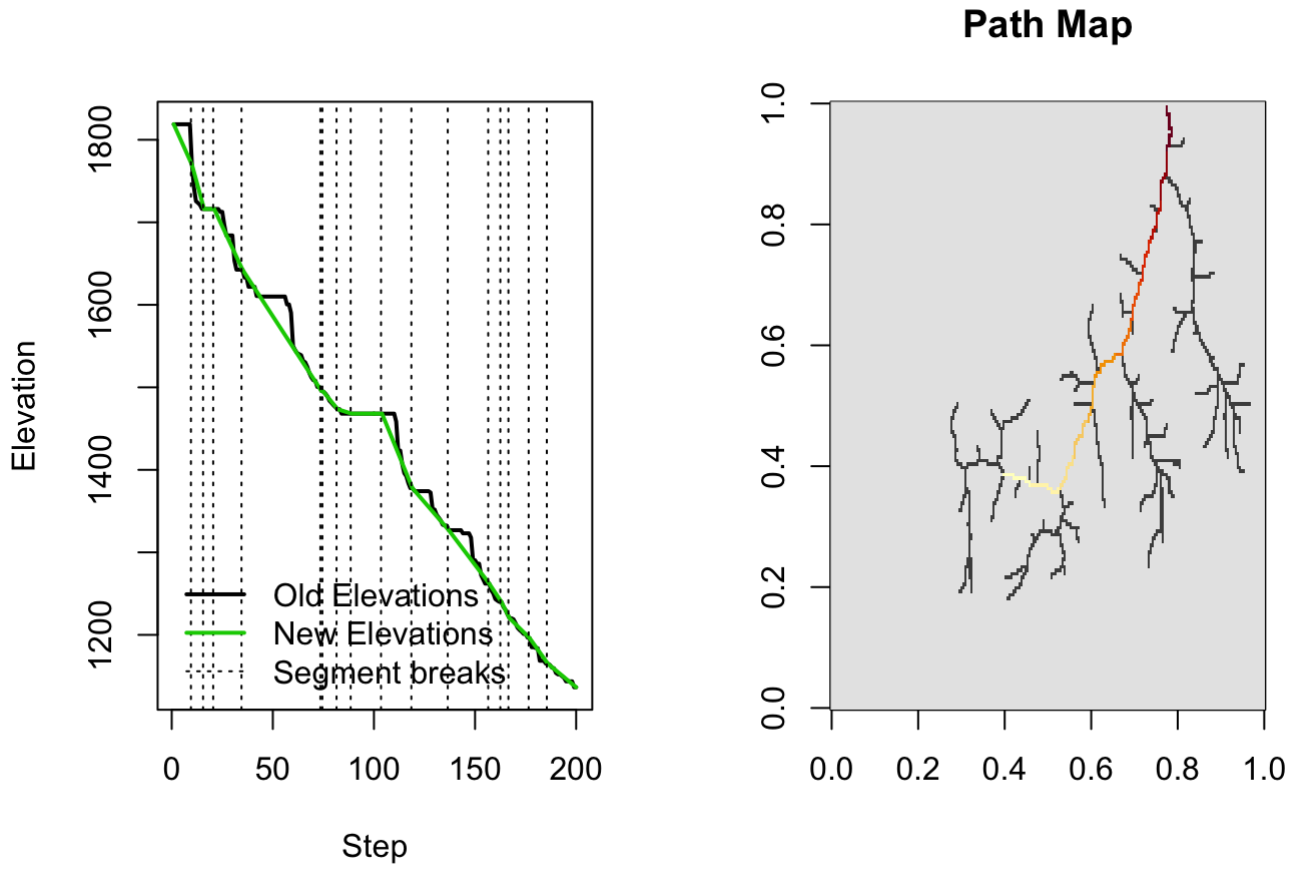
```
segment=30 #pick a stream segment to walk down from, change this number to see a differe
nt segment

#Use PathExtract to extract outputs along a flowpath
start=c(subbasin$summary[segment,2], subbasin$summary[segment,3])
streamline_old=PathExtract(input=travHS$dem, direction=travHS$direction, mask=watershed.
mask, startpoint=start)
streamline_new=PathExtract(input=RivSmooth$dem.adj, direction=travHS$direction, mask=wat
ershed.mask, startpoint=start)
streamline_riv=PathExtract(input=subbasin$segments, direction=travHS$direction, mask=wat
ershed.mask, startpoint=start)

transect_old=streamline_old$data
transect_new=streamline_new$data
transect_riv=streamline_riv$data

#plot the path
#quartz()
par(mfrow=c(1,2))
  #plot the elevation transects
  nstep=length(transect_riv)
  slist=which(transect_riv[2:nstep]!=transect_riv[1:(nstep-1)])+0.5 #find the breaks bet
ween segments
  limit=range(c(transect_old, transect_new))
  plot(1:nstep, transect_old, ylim=c(limit[1],limit[2]), col=1, lwd=2, type='l', xlab="S
tep", ylab="Elevation")
  lines(1:nstep, transect_new, lty=1, lwd=2, col=3)
  abline(v=slist, col=1, lty=3)
  legend("bottomleft", c("Old Elevations", "New Elevations", "Segment breaks"), bty='n',
lty=c(1,1,3), col=c(1,3,1), lwd=c(2,2,1))

  #plot the path
  segment.plot=subbasin$segments
  segment.plot[segment.plot>0]=1
  image(segment.plot, breaks=c(0,0.5,1.5), col=rev(gray.colors(2)), add=F, main="Path Ma
p")
  plot.path=streamline_riv$path.mask
  plot.path[plot.path==0]=NA
  image(plot.path, add=T)
```



Step 3: Calculate the slopes

Once you have the DEM how you want it you can calculate the slopes for use in ParFlow. Here I am using the *SlopeCalcStan* function which does not do any downwinding and is intended for use with the **OverlandKinematic** or **OverlandDiffusive** boundary conditions in ParFlow. If you are using the **OverlandFlow** boundary condition you should use the *SlopeCalcUP* function documented in the Downwinding_Workflow_Examples.

The mandatory inputs for this function are:

1. The processed DEM that you would like to calculate the slopes from
2. The flow direction file from *Step 1*
3. The resolution of the DEM: dx, dy (this should be in the same units as your elevations are reported in)

The following optional inputs are also being used here (refer to the help("SlopeCalcStan) for more details):

1. minslope - This ensures that the magnitude of all slopes is greater than or equal to this value along the flow directions given in the flow direction file
2. maxslope - This sets a global maximum slope magnitude
3. Secondary TH - If this is set to 0 then all slopes not along the primary flow directions (i.e. the x slope for a cell with a primary direction up or down, or the y slope for a cell with a primary direction to the left or right) will be set to zero. If this is set to the default -1 value no adjustments will be made

```
slopesCalc=SlopeCalStan(dem=RivSmooth$dem.adj, mask=watershed.mask, direction=travHS$dir
ection, minslope=1e-5, maxslope=1, dx=1000, dy=1000, secondaryTH=-1)
```

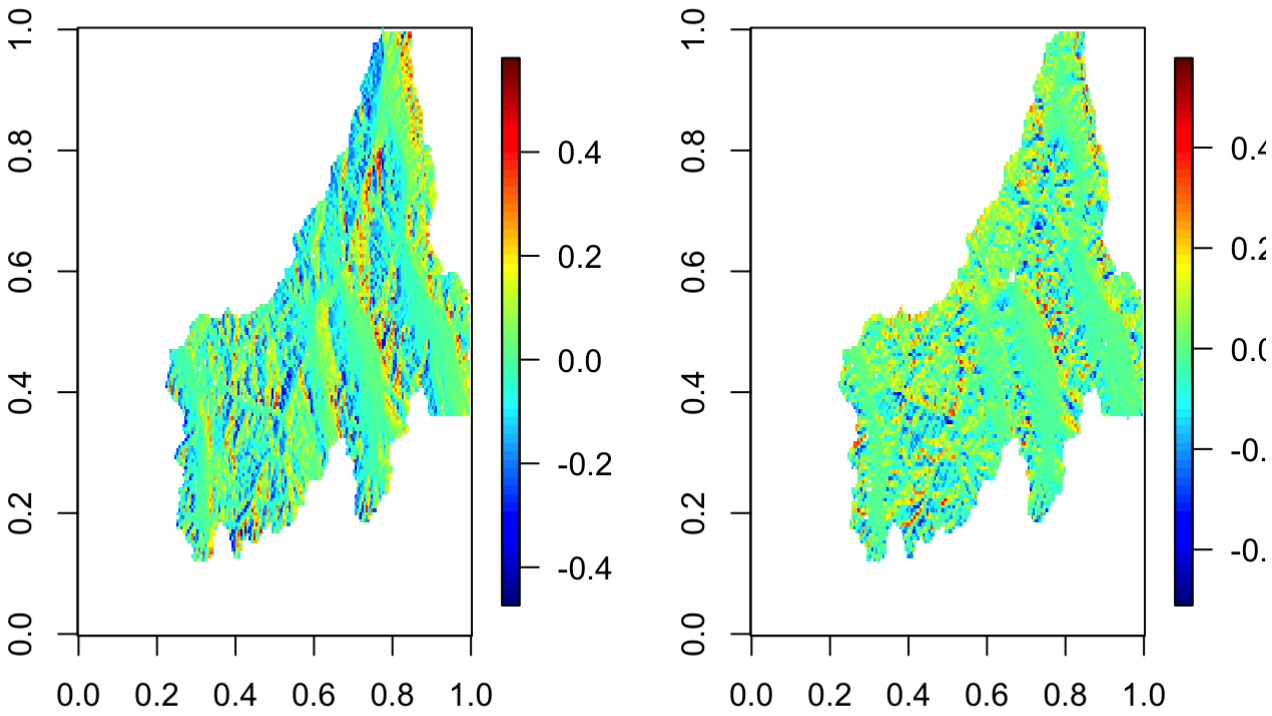
```
## [1] "Limiting slopes to minimum 1e-05"
## [1] "Limiting slopes to maximum absolute value of 1"
```

Plotting the resulting slopes

```
slopex=slopesCalc$slopex
slopey=slopesCalc$slopey

sxplot=slopex
sxplot[which(slopex==0)]=NA
syplot=slopey
syplot[which(slopey==0)]=NA

#plot
par(mfrow=c(1,2))
image.plot(sxplot)
image.plot(syplot)
```



Step 4: Write slope files out in ParFlow ascii format

Any of the matrices calculated here can be written out in any user preferred format. Here I just provide an example of how to write out the slope files in the standard format needed for ParFlow. This ASCII file can be converted to pfb or silo using standard pftools.

```
nx=dim(slopex)[1]
ny=dim(slopex)[2]
slopexPF=slopeyPF=DEMPF=DIRPF=rep(0, nx*ny)
jj=1
for(j in 1:ny){
  for(i in 1:nx){
    slopexPF[jj]=slopex[i,j]
    slopeyPF[jj]=slopey[i,j]
    DEMPF[jj]=RivSmooth$dem.adj[i,j]
    DIRPF[jj]=travHS$direction[i,j]
    jj=jj+1
  }
}

#write statements commented out for the example uncomment to write the files
#write.table( t(c(nx,ny,1)), "SlopeX.sa", append=F, row.names=F, col.names=F)
#write.table(slopexPF, fout, append=T, row.names=F, col.names=F)

#write.table( t(c(nx,ny,1)), "SlopeY.sa", append=F, row.names=F, col.names=F)
#write.table(slopeyPF, fout, append=T, row.names=F, col.names=F)

#write.table( t(c(nx,ny,1)), "DEM.Processed.sa", append=F, row.names=F, col.names=F)
#write.table(DEMPF, fout, append=T, row.names=F, col.names=F)

#write.table( t(c(nx,ny,1)), "Flow.Direction.sa", append=F, row.names=F, col.names=F)
#write.table(DIRPF, fout, append=T, row.names=F, col.names=F)
```