Venus Craters

Andus Kong

Friday, November 13, 2015

1.

```r
data <- read.delim(file = "http://www.lpi.usra.edu/venus/craters/rel3main.txt")
# Clean Data
column_names <- as.character(unlist(data[2,]))
final_data <- data[3:nrow(data),]
colnames(final_data) <- column_names
```

2.

```r
library("geosphere")
```

```
## Loading required package: sp
```

```r
VenusR <- 6051.8
######
# Calculates Haversine Distance
#
# @param x: longitude/latitude of point(s).
#           Can be a vector of two numbers, a matrix of 2 columns
# @param y: as above
# @param r: Radius; default = VenusR or 6051.8 km: radius of Venus
# return:   Haversine Distance
######
dis <- function(x, y, r = VenusR){
  Haversine_d <- distHaversine(x, y, r)
  return(Haversine_d)
}
```

3.

```r
library("sphereplot")
```

```
## Loading required package: rgl
```

```r
# The following packages allow the output to be visible in an online browser
library(knitr)
library(rgl)
knit_hooks$set(webgl = hook_webgl)

longitude <- as.numeric(as.character(final_data$Lon))
latitude <- as.numeric(as.character(final_data$Lat))
radius <- rep(VenusR, length(longitude))
coordinates <- cbind(longitude, latitude, radius)
rgl.sphgrid(radius = VenusR, radaxis = FALSE,deggap = 45, longtype = "D")
# if you open in browser i.e. Google Chrome, you will see the plot
rgl.sphpoints(long = coordinates)
```

You must enable Javascript to view this page properly.

4.

a.

```
######
# Calculates the weighted mean of the elevations
#
# @param point: a vector of two numbers - longitude and latitude
# @param coordinates: Three column matrix - lon, lat, elevation respectively
# @param R: Radius; default = 6051.8 km, radius of Venus
# @param kernel: Specifies the spatial kernel smoothing method;
#                default is "Gaussian". Possible inputs:"Uniform","Epanechnikov"
#                "Triangular"
# @param b: bandwidth; default is 13000 for Venus elevation testing purposes
# return: weighted mean of the elevations
######
Spatial_Kernel_Smoother <- function(point, coordinates, R = VenusR, kernel = "Gaussian", b = 13000){

  no_elevation_coordinates <- coordinates[,1:2]
  distance <- vector()
  elevations <- coordinates[,3]

  # Create distance vector
  for(i in 1:nrow(no_elevation_coordinates)){

     distance[i] <- dis(point, no_elevation_coordinates[i,], R)

  }

  # Calculate Gaussian weights
  if (kernel == "Gaussian"){

    weights <- vector()

    for (i in 1:nrow(no_elevation_coordinates)){

      weights[i] <- exp(1) ^ (-distance[i]^2 / (2 * b^2))
    }
  }

  # Calculate Uniform weights
  if (kernel == "Uniform"){
    weights <- vector()

    for(i in 1:nrow(no_elevation_coordinates)){
      if (distance[i] <= b){
        weights[i] <- distance[i]
      }
      else{
        weights[i] <- 0
      }
```

```
    }


  }
  # Calculate Epanechnikov weights
  if (kernel == "Epanechnikov"){
    weights <- vector()

    for(i in 1:nrow(no_elevation_coordinates)){
      if (distance[i] <= b){
      weights[i] <- ((b^2 - distance[i]^2) * distance[i])
      }
      else{
      weights[i] <- 0
      }
    }

  }

  # Calculate Triangular weights
  if (kernel == "Triangular"){
    weights <- vector()
    for(i in 1:nrow(no_elevation_coordinates)){
      if (distance[i] <= b){
      weights[i] <- ((b - distance[i]) * distance[i])
      }
      else{
      weights[i] <- 0
      }
    }
  }
  product <- sum(weights * elevations)
  weighted_mean <- product / sum(weights)
  return(weighted_mean)

}
```

b.

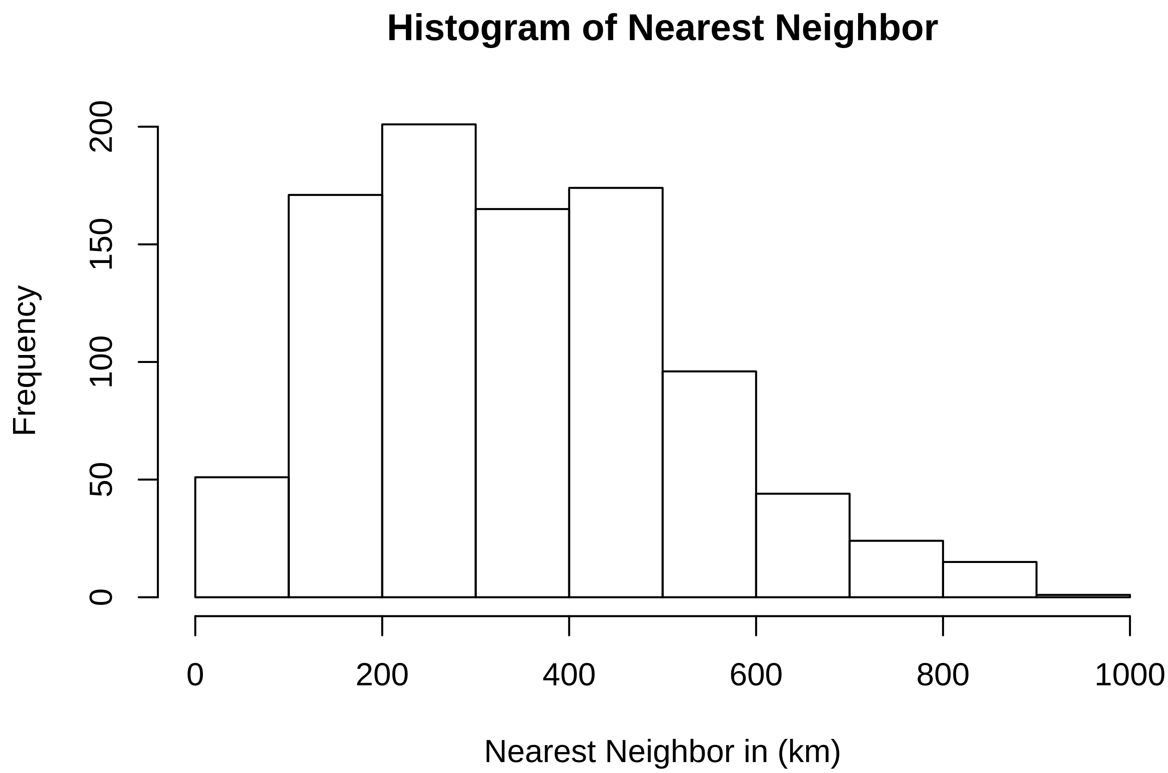
```
# Create necessary dataframes
elevations <- as.numeric(as.character(final_data$Ev))
coordinates <- cbind(longitude, latitude, elevations)
index <- which(!is.na(coordinates[,3]))
missing_elevations <- coordinates[-index,]
missing_elevations_coordinates <- missing_elevations[,1:2]
coordinates <- coordinates[index,]

# Run through every kernel method
Gaus_elevations_estimates <- vector()
for(i in 1:nrow(missing_elevations_coordinates)){
  Gaus_elevations_estimates[i] <- Spatial_Kernel_Smoother(missing_elevations_coordinates[i,], coordinat
}
```

```r
Epanc_elevations_estimates <- vector()
for(i in 1:nrow(missing_elevations_coordinates)){
  Epanc_elevations_estimates[i] <- Spatial_Kernel_Smoother(missing_elevations_coordinates[i,], coordina
}

Tri_elevations_estimates <- vector()
for(i in 1:nrow(missing_elevations_coordinates)){
  Tri_elevations_estimates[i] <- Spatial_Kernel_Smoother(missing_elevations_coordinates[i,], coordinate
}

Uni_elevations_estimates <- vector()
for(i in 1:nrow(missing_elevations_coordinates)){
  Uni_elevations_estimates[i] <- Spatial_Kernel_Smoother(missing_elevations_coordinates[i,], coordinate
}

predicted <- data.frame(missing_elevations_coordinates,Gaus_elevations_estimates, Epanc_elevations_esti
predicted
```

```
##   longitude latitude Gaus_elevations_estimates Epanc_elevations_estimates
## 1     357.7    -82.5                  6051.843                   6051.933
## 2     337.7     86.8                  6051.856                   6051.883
## 3     142.1    -72.9                  6051.846                   6051.958
## 4     145.0    -73.0                  6051.846                   6051.960
## 5     259.0     87.3                  6051.856                   6051.898
## 6     105.5     80.6                  6051.862                   6051.937
## 7     109.0     77.4                  6051.863                   6051.948
##   Tri_elevations_estimates Uni_elevations_estimates
## 1                 6051.905                 6051.937
## 2                 6051.861                 6051.998
## 3                 6051.950                 6051.869
## 4                 6051.951                 6051.872
## 5                 6051.877                 6051.994
## 6                 6051.916                 6051.976
## 7                 6051.929                 6051.971
```

c.

```r
coordinates <- cbind(longitude, latitude)
nearest_neighbor <- vector()
# calculates nearest neighbor through min function
for (i in 1:nrow(coordinates)){

  nearest_neighbor[i] <- min(dis(coordinates[i,], coordinates[-i,], r = VenusR))

}

hist(nearest_neighbor, xlab = "Nearest Neighbor in (km)", main = "Histogram of Nearest Neighbor")
```



5.

a.

```
######
# Calculates Ripley's K function values
#
# @param spatial_points: matrix of longitude and latitude points
# @param R: radius
# @param Dm: vector of distances; default is the required vector in problem
# returns: vector of Ripley's K function values
######
K_function <- function(spatial_points, R, Dm = seq(1,100) * pi * VenusR / 100){
  # plyr parallelizes the respective matrix computations
  require(plyr)
  k_values <- vector()
  n <- nrow(spatial_points)
  distance <- list()

  constant <- (4 * pi * R^2) / (n * (n - 1))

  # Create list of lists of distances of each respective point
  for(i in 1:n){

    distance[[i]] <- dis(spatial_points[i,], spatial_points[-i,], R)
```

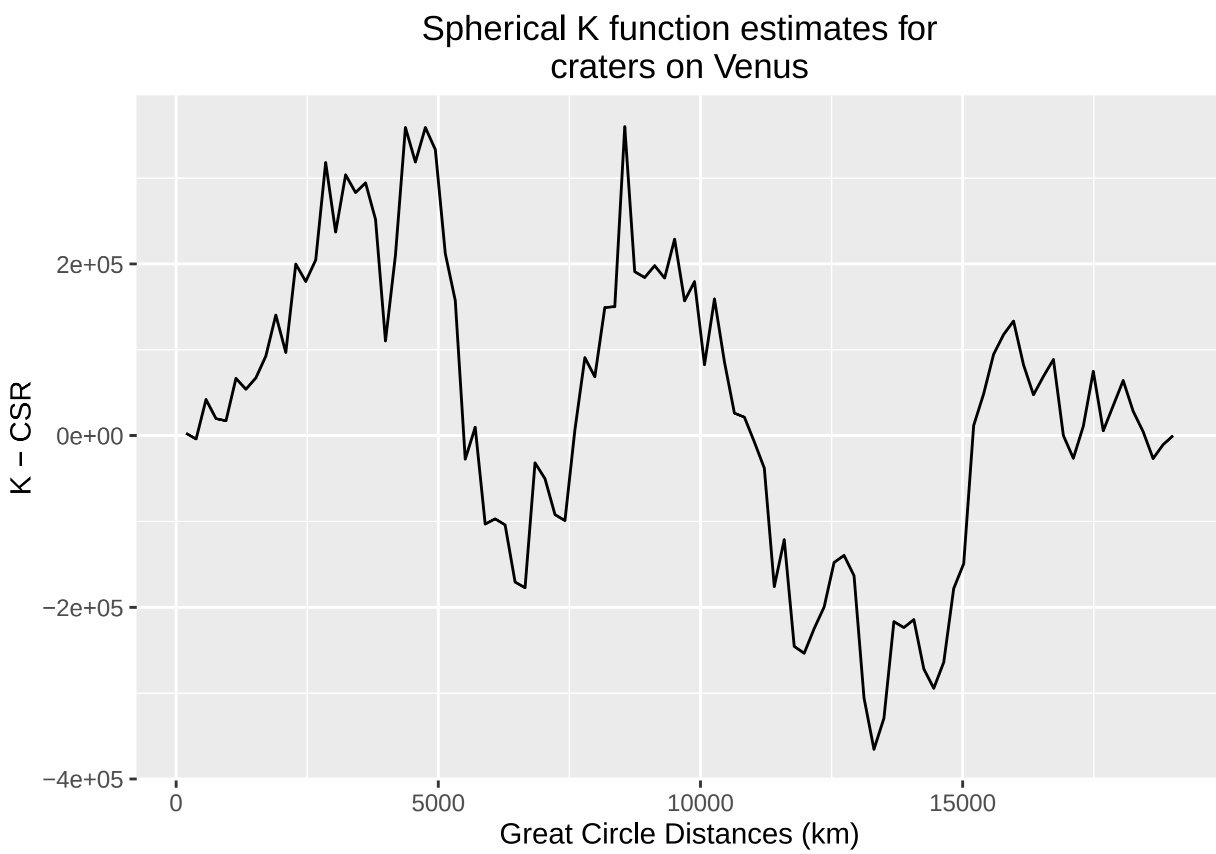
```
  }

  # Find where distances are less than Dm
  index <- list()
  index <- lapply(Dm, FUN = function(x) distance[[i]] <= x)
  summation <- lapply(index, sum)
  summation <- unlist(summation)

  for (i in 2:n){
    index <- lapply(Dm, FUN = function(x) distance[[i]] <= x)
    temp <- lapply(index, sum)
    x <- data.frame(summation, unlist(temp))
    summation <- apply(x, 1, FUN = sum)
  }

  summation <- constant * summation
  return(summation)
}
```

b.

```
# Calculates and plots difference in Ripley's_K and CSR_K
longitude <- as.numeric(as.character(final_data$Lon))
latitude <- as.numeric(as.character(final_data$Lat))
coordinates <- cbind(longitude, latitude)
Dm = seq(1,100) * pi * VenusR / 100
Ripleys_K <- K_function(coordinates, VenusR)
```

```
## Loading required package: plyr
```

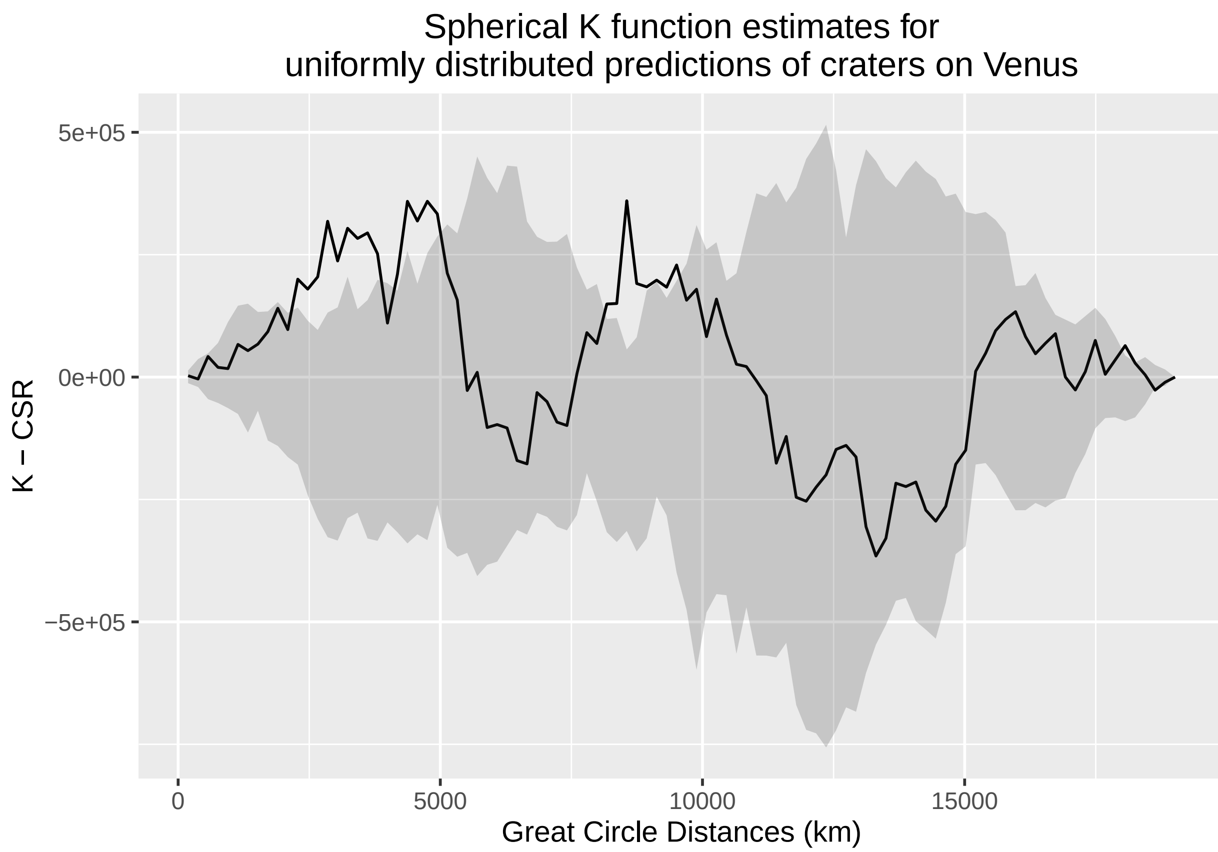
```
CSR_K <- (2 * pi * VenusR^2) * (1- cos(Dm / VenusR))
library("ggplot2")
  p1 <- ggplot() + geom_line(aes(Dm, Ripleys_K - CSR_K)) +
    labs(title = "Spherical K function estimates for\ncraters on Venus",
       x = "Great Circle Distances (km)",
       y = "K - CSR")
  print(p1)
```



6.

```
######
# Outputs Venus K Value difference plots and plots upper and lower bound for each dm
#
# @param Ripleys_K: Calculated K values for Venus spatial points
# @param n: Amount of times to simulate distribution; default 100
# @ return: Outputs the plot of the lb and ub of each dm
######
Uniform_Venus <- function(Ripleys_K, n = 100){

  R <- vector()
  lon <- vector()
  lat <- vector()
  Sample_K <- matrix(nrow = 100, ncol = n)

  # Generate Uniform Distributions
  for (i in 1:n){
    R <- runif(942, min = -VenusR, max = VenusR)
    lon <- runif(942, min = 0, max = 360)
    lat <- asin(R / VenusR) * 180 / pi
    coordinates <- data.frame(lon,lat)
    Sample_K[,i] <- K_function(coordinates, VenusR)
  }
  # Build dataframe
```

```
  bounds <- apply(Sample_K, MARGIN = 1,FUN = function(x) quantile(x, c(0.05,0.95)))
  lb <- bounds[1,]
  ub <- bounds[2,]
  Dm = seq(1,100) * pi * VenusR / 100
  CSR_K <- (2 * pi * VenusR^2) * (1- cos(Dm / VenusR))

  # Plot lower and upper bounds for each dm
  p1 <- ggplot() + geom_line(aes(Dm, Ripleys_K - CSR_K)) +
    geom_ribbon(aes(x = Dm, ymin = lb - CSR_K, ymax = ub - CSR_K), alpha = 0.2) +
    labs(title = "Spherical K function estimates for\nuniformly distributed predictions of craters on Ve
       x = "Great Circle Distances (km)",
       y = "K - CSR")
  print(p1)
}

# For performance purposes, I used 10 to demonstrate the ability of the
# function to plot the lb and ub for each dm.
# Inputting 100 into n will allow for 100 samples
Uniform_Venus(Ripleys_K, n = 10)
```



I conclude that the craters on Venus are randomly distributed in a uniform distribution across Venus. The actual sample k values did not differ too far from the upper and lower bound. Also the uniformly simulated graph closely resembles the actual graph of data gathered from Venus.