Demo#4-Kriging

Omnidirectional Kriging

Jeffrey Yarus

08 juillet, 2025

Contents

1 Demo Summary

In this demonstration, the objective is to create a Kriged map and it's associated error variance map. Here, the code begins as usual by reading in the data set, creating the assignment variables, identifying outliers, and removing the outliers. We are now ready to create variograms. **Please be sure to read through the text I have included before each code chunk.**

Run the entire code chunk by chunk. The variable P_PHI_pct is already in place and ready to run. The variogram model and kriging parameters are already set. **Erase your Global Environment before beginning**

Perform Omnidirectional kriging of P__PHI__pct For variogram modeling the best models to use are the: - Exponential - Spherical - Cubic - Stable - Be sure the data base you select has omitted any outliers

Note which variogram model types generated the "best results" for omnidirectional kriging (just through visual inspection, you don't have to do bootstrapping)

2 Terminology and key functions from this Demo

1. **Kriging**
   - The geostatistical interpolation method
2. **Unique Neighborhood**
   - Kriging requires knowledge of the neighbors it will use for estimating the values at each grid cell. The default parameter is called a, "Unique" neighborhood which instructs kriging to use all the data in the study area. In this case, the number of samples is considered small, so we can use all the data. In other cases, you will want to establish a moving neighborhood which restricts the number of samples to use in the estimation around a grid cell.
3. **Kriging Estimate**
   - The value determined from the kriging equation at a given location. The estimage is optimized and considered the, "Best Linear, Unbiased Estimate.
4. **Error Estimate**
   - The value of the variance or standard deviation determined from the kriging equation at a given location. The Error Variance is usually reported in terms of the standard deviation as the standard deviation is in the same units as the original variable.

3 Loading Packages

Run the chunk below…

4 Initializing the Demo

As in the previous demos, we:

- Read in our data
- Create the tibble
- Ensure our categorical variables are designated as factors
- Create a backup data frame
- Remove the unnecessary variables from the tibble
- Create our assigned symbolic names
- Identify the outliers. all of this is being done in one code chunk

Run the chunk below…

```r
file.name <- "../../data/WT-2D-all-outlier.csv"
df <- read_csv(file.name)
```

```
## Rows: 262 Columns: 11
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): F_Top
## dbl (10): N_Well, C_X_ft, C_Y_ft, L_3_FACIES, P_Delta_t, P_KH_md, P_PHI, P_P...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
df %<>%
  mutate(across(matches("N_|L_"), factor))
head(df, n = 20)
```

```
## # A tibble: 20 x 11
##    F_Top       N_Well C_X_ft C_Y_ft L_3_FACIES P_Delta_t P_KH_md P_PHI P_PHI_pct
##    <chr>       <fct>   <dbl>  <dbl> <fct>          <dbl>   <dbl> <dbl>     <dbl>
##  1 S_Siltstone 5001    11564   5691 2               60.8    3.51 0.094       9.4
##  2 S_Siltstone 5002    10679  13706 2               59.1    5.81 0.093       9.3
##  3 S_Siltstone 5003     6311  36307 1               55.9    0.77 0.059       5.9
##  4 S_Siltstone 5004     2754  11437 1               52.5    0.55 0.044       4.4
##  5 S_Siltstone 5005     9386  24799 3               63.3    6.29 0.099       9.9
##  6 S_Siltstone 5006    10761  24755 3               63.1    2.9  0.099       9.9
##  7 S_Siltstone 5007    10769  23463 3               65.9   12.3  0.113      11.3
##  8 S_Siltstone 5008     9145  23413 1               57.9    0.79 0.074       7.4
##  9 S_Siltstone 5009     8201  23557 3               64.4    8.19 0.104      10.4
## 10 S_Siltstone 5010     6727  23472 1               55.8    1.45 0.071       7.1
## 11 S_Siltstone 5011     3834  23446 1               53.9    1.6  0.062       6.2
## 12 S_Siltstone 5012     2780  23494 1               54.8    1.57 0.063       6.3
## 13 S_Siltstone 5013     3994  22098 1               53.5    1.51 0.06        6
## 14 S_Siltstone 5014     5482  22214 1               53.9    1.79 0.065       6.5
## 15 S_Siltstone 5015     6692  22138 2               60.8    8.21 0.091       9.1
## 16 S_Siltstone 5016     9324  22199 3               63.5    4.48 0.1        10
## 17 S_Siltstone 5017    10817  22305 2               61.7    2.73 0.093       9.3
## 18 S_Siltstone 5018    10761  20774 1               57.4    2.68 0.073       7.3
## 19 S_Siltstone 5019     9350  20761 3               62.5    3.42 0.096       9.6
## 20 S_Siltstone 5020     8237  20748 1               57.2    3.82 0.079       7.9
## # i 2 more variables: P_Thickness <dbl>, P_Top_ft <dbl>
```

```r
# Creating a duplicate tibble for later use, but eliminate unnecessary variables

df2 <- df %>%
  dplyr::select(L_3_FACIES:P_Top_ft)

# Creating Symbolic Inputs

xlon <- "C_X_ft"
ylat <- "C_Y_ft"
out_analysis <- "Raw (outlier analysis performed on raw data)"
property <- "P_PHI_pct" # The property you are selecting for analysis

# Idnetifying the outliers

df <-
  df %>%
  mutate(
    iqr_val = IQR(!!sym(property)),
    # calculates the IQR value
    iqr_val_adj = ((iqr_val) * 1.5),
    third_q = quantile(!!sym(property), prob = 0.75, na.rm = TRUE),
    first_q = quantile(!!sym(property), prob = .25, na.rm = TRUE),
    # Creating a column of True and False, True = an outlier, False = not an
    #outlier
    outlier =
      (!!sym(property)) > (third_q + iqr_val_adj) |
      (!!sym(property) < (first_q - iqr_val_adj))
  )

df
```

```
## # A tibble: 262 x 16
##    F_Top       N_Well C_X_ft C_Y_ft L_3_FACIES P_Delta_t P_KH_md P_PHI P_PHI_pct
##    <chr>       <fct>   <dbl>  <dbl> <fct>          <dbl>   <dbl> <dbl>     <dbl>
##  1 S_Siltstone 5001    11564   5691 2               60.8    3.51 0.094       9.4
##  2 S_Siltstone 5002    10679  13706 2               59.1    5.81 0.093       9.3
##  3 S_Siltstone 5003     6311  36307 1               55.9    0.77 0.059       5.9
##  4 S_Siltstone 5004     2754  11437 1               52.5    0.55 0.044       4.4
##  5 S_Siltstone 5005     9386  24799 3               63.3    6.29 0.099       9.9
##  6 S_Siltstone 5006    10761  24755 3               63.1    2.9  0.099       9.9
##  7 S_Siltstone 5007    10769  23463 3               65.9   12.3  0.113      11.3
##  8 S_Siltstone 5008     9145  23413 1               57.9    0.79 0.074       7.4
##  9 S_Siltstone 5009     8201  23557 3               64.4    8.19 0.104      10.4
## 10 S_Siltstone 5010     6727  23472 1               55.8    1.45 0.071       7.1
## # i 252 more rows
## # i 7 more variables: P_Thickness <dbl>, P_Top_ft <dbl>, iqr_val <dbl>,
## #   iqr_val_adj <dbl>, third_q <dbl>, first_q <dbl>, outlier <lgl>
```

4.1 Code chucks that apply to lognormal data only

The variable, P_PHI_pct is roughly symetrically distributed. If you decide to try a different variable, you need to check to see if it is symetrically distributed. If it is not, and the variable data are symmetrically distributed YOU MAY SKIP THE NEXT TWO CHUNKS!!!

Adding the log of a right-skewed (~normally distributed) variable This property will be added to the data frame for the purpose of identifying the outlier. However, it's only useful if the variable being evaluated is log-normally distributed. should you decide to change the variable under investigation from P_PHI_pct to P_KH_md, then this section becomes relevant.

ADJUSUTING FOR THE MIN AND MAX WITHOUT THE OUTLIER Once the outlier is found, in order to identify the proper interquartile range (without the outlier), we need to redefine the max and min values. Note, again, this is operating on the Log_property in case the variable under investigation is log-normally distributed.

Run the code chunk below **if the variable you have selected is log-nomrally distributed, uncomment the code chunk below and run it. This will take the log of the variable temporarily just to correctly identify outliers.**

```
# df <-
#   df %>%
#   mutate(Log_property = log(!!sym(property)))
#
# df <-
#   df %>%
#   mutate(
#     maxval_trans = max(Log_property),
#     # not necessary to calculate max
#     minval_trans = min(Log_property),
#     # not necessary to calculate min
#     iqr_val_trans = IQR(Log_property),
#     # calculates the IQR value
#     iqr_val_adj_trans = ((iqr_val_trans) * 1.5),
#     third_q_trans = quantile(Log_property, prob = 0.75, na.rm = TRUE),
#     first_q_trans = quantile(Log_property, prob = .25, na.rm = TRUE),
#     outlier =
#       (Log_property) > (third_q_trans + iqr_val_adj_trans) |
#       (Log_property) < (first_q_trans - iqr_val_adj_trans)
#   ) %>%
#   dplyr::select(-maxval_trans:-first_q_trans)
```

4.2 End of code chunks for log-normally distributed data data

5 Working in R S4 - Preparation for Geostatistical Analysis using gstlearn

5.1 Creating the Databases and Basemap

The following chunk creates two databases, one **with outliers,** and one **without outliers** from a data frame and assigns the location of the coordinates and the selected mapping variable ("property"). Then, the basemap with no outlier is plotted for reference.

Run the code chunk below...

6 creating databases with and without the outlier

```
# creating a database with the outlier
df1 = df %>% dplyr::select(-F_Top)

# creating a database with the outlier from the Data Frame
db = Db_fromTL(df1)

# Define the different variables
err = db$setLocators(c(xlon, ylat), ELoc_X())
err = db$setLocator(property, ELoc_Z())
db
```

```
##
## Data Base Characteristics
## =========================
##
## Data Base Summary
## -----------------
## File is organized as a set of isolated points
## Space dimension              = 2
## Number of Columns            = 15
## Total number of samples      = 262
##
```

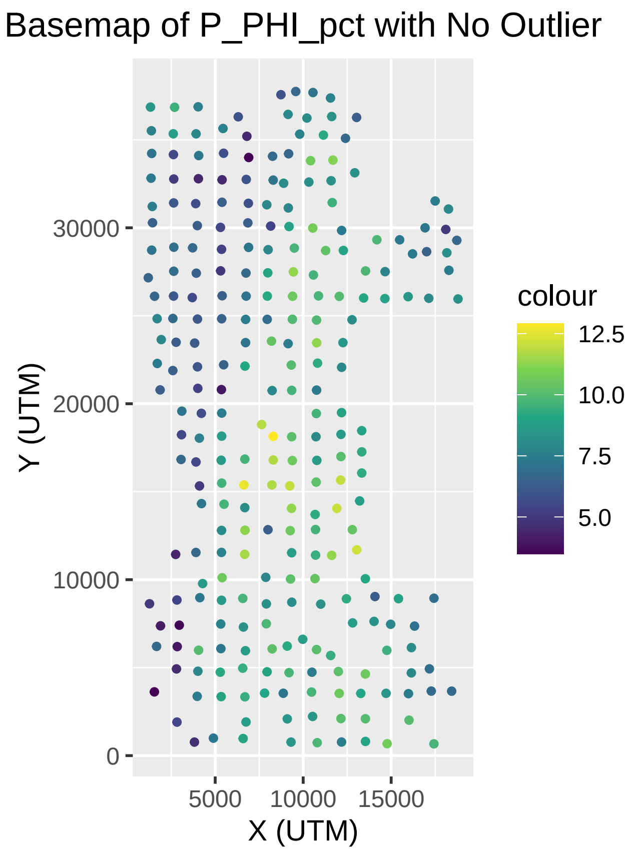
```
## Variables
## ---------
## Column = 0 - Name = N_Well - Locator = NA
## Column = 1 - Name = C_X_ft - Locator = x1
## Column = 2 - Name = C_Y_ft - Locator = x2
## Column = 3 - Name = L_3_FACIES - Locator = NA
## Column = 4 - Name = P_Delta_t - Locator = NA
## Column = 5 - Name = P_KH_md - Locator = NA
## Column = 6 - Name = P_PHI - Locator = NA
## Column = 7 - Name = P_PHI_pct - Locator = z1
## Column = 8 - Name = P_Thickness - Locator = NA
## Column = 9 - Name = P_Top_ft - Locator = NA
## Column = 10 - Name = iqr_val - Locator = NA
## Column = 11 - Name = iqr_val_adj - Locator = NA
## Column = 12 - Name = third_q - Locator = NA
## Column = 13 - Name = first_q - Locator = NA
## Column = 14 - Name = outlier - Locator = NA
```

```r
# Creating databases with no outliers
# Remove the first variable
df1 = df %>% dplyr::select(-F_Top) %>% filter(outlier == FALSE)

# creating a database with the outlier from the Data Frame
db_noout.db = Db_fromTL(df1)

# Define the different variables
err = db_noout.db$setLocators(c(xlon, ylat), ELoc_X())
err = db_noout.db$setLocator(property, ELoc_Z())
db_noout.db
```

```
##
## Data Base Characteristics
## =========================
##
## Data Base Summary
## -----------------
## File is organized as a set of isolated points
## Space dimension              = 2
## Number of Columns            = 15
## Total number of samples      = 261
##
## Variables
## ---------
## Column = 0 - Name = N_Well - Locator = NA
## Column = 1 - Name = C_X_ft - Locator = x1
## Column = 2 - Name = C_Y_ft - Locator = x2
## Column = 3 - Name = L_3_FACIES - Locator = NA
## Column = 4 - Name = P_Delta_t - Locator = NA
## Column = 5 - Name = P_KH_md - Locator = NA
## Column = 6 - Name = P_PHI - Locator = NA
## Column = 7 - Name = P_PHI_pct - Locator = z1
## Column = 8 - Name = P_Thickness - Locator = NA
## Column = 9 - Name = P_Top_ft - Locator = NA
## Column = 10 - Name = iqr_val - Locator = NA
## Column = 11 - Name = iqr_val_adj - Locator = NA
## Column = 12 - Name = third_q - Locator = NA
## Column = 13 - Name = first_q - Locator = NA
## Column = 14 - Name = outlier - Locator = NA
```

Plot the property of interest from the data base

```
p = plot.init(asp=1)
p = p + plot.symbol(db_noout.db, nameColor = property, pch = 19, cex = 1, flagLegend = TRUE)
p = p + plot.decoration(title=paste("Basemap of", property, "with No Outlier"), xlab = "X (UTM)", ylab =
plot.end(p)
```



To determine the ceiling of the grid (or the coordinates of the upper-right corner), I wrote a roundUp() function that take the largest value in both selected C_X and C_Y column and find the nearest 10, 100, 1000, etc. that's larger than the largest value. The purpose of this code is to establish the extents of the data so we can build a grid.

7 Variogram Construction and modeling

7.1 Omnidirectional Experimental Semivariogram

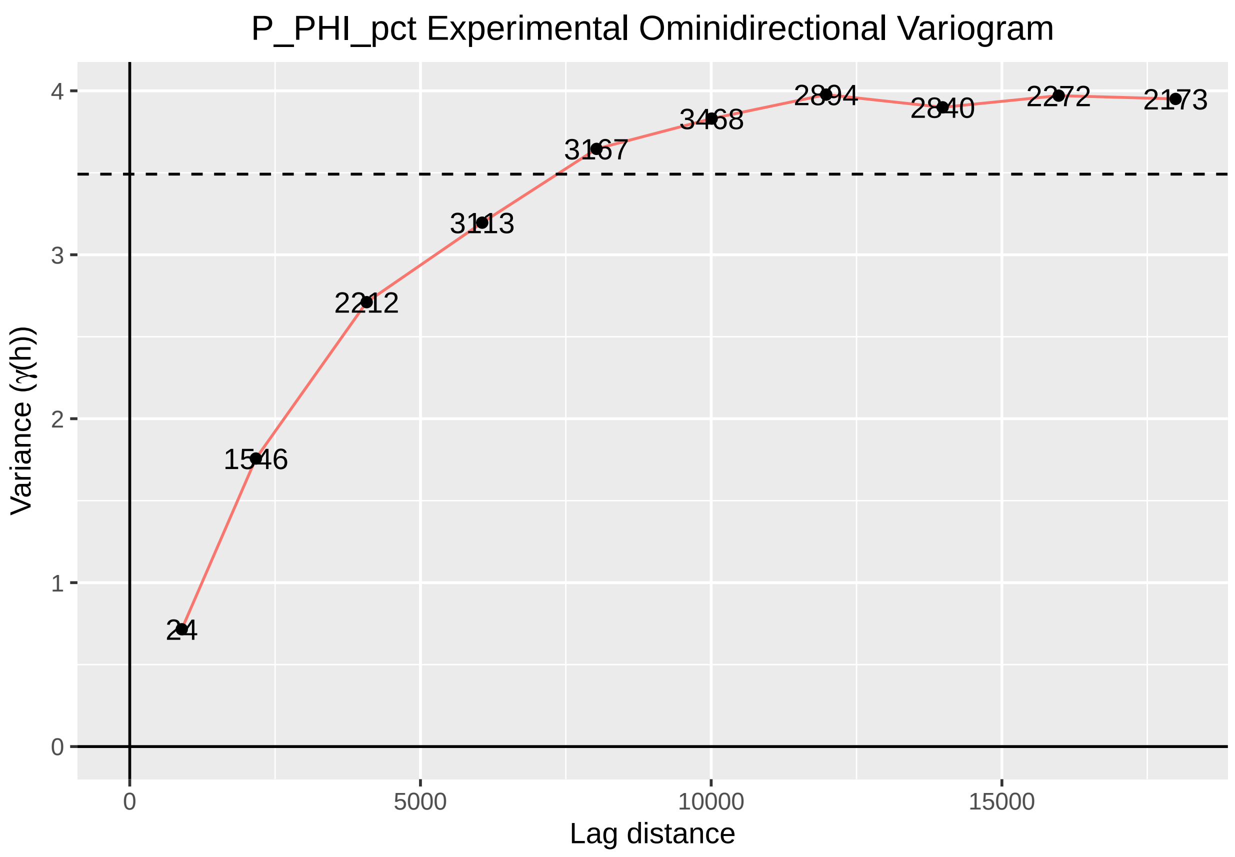
The code chunks below calculate and plot the variograms. Initially, they do not correct for outliers. Replace the database (db) with the database that removes the outliers (db_noout.db). Observe the impact.

Run the chunk below…

```
varioparam <- VarioParam_createOmniDirection(nlag = 10, dlag=2000)
vario_omni <- Vario_computeFromDb(varioparam, db_noout.db)
```

Represent the variogram

```
p = plot.init()
p = p + plot.vario(vario_omni, drawPlabel = TRUE)
p = p + plot.decoration(title = paste(property, "Experimental Ominidirectional Variogram"),
                        xlab = "Lag distance",
                        ylab = expression(paste("Variance (", gamma, "(h))", sep="")))
plot.end(p)
```



7.2 Modeling the Experimental Semivariogram

8 Selecting the variogram model(s)

Run the chunk below…

```
ECov_printAll()
```

```
##   -2 -     UNKNOWN : Unknown covariance
##   -1 -    FUNCTION : External covariance function
##    0 -      NUGGET : Nugget effect
##    1 - EXPONENTIAL : Exponential
##    2 -   SPHERICAL : Spherical
##    3 -    GAUSSIAN : Gaussian
##    4 -       CUBIC : Cubic
##    5 -     SINCARD : Sine Cardinal
##    6 -     BESSELJ : Bessel J
##    7 -      MATERN : Matern
##    8 -       GAMMA : Gamma
##    9 -      CAUCHY : Cauchy
##   10 -      STABLE : Stable
##   11 -      LINEAR : Linear
##   12 -       POWER : Power
##   13 -   ORDER1_GC : First Order Generalized covariance
##   14 -   SPLINE_GC : Spline Generalized covariance
##   15 -   ORDER3_GC : Third Order Generalized covariance
##   16 -   ORDER5_GC : Fifth Order Generalized covariance
##   17 -     COSINUS : Cosine
##   18 -    TRIANGLE : Triangle
##   19 -      COSEXP : Cosine Exponential
##   20 -       REG1D : 1-D Regular
##   21 -       PENTA : Pentamodel
##   22 -  SPLINE2_GC : Order-2 Spline
##   23 -     STORKEY : Storkey covariance in 1-D
##   24 -   WENDLAND0 : Wendland covariance (2,0)
##   25 -   WENDLAND1 : Wendland covariance (3,1)
##   26 -   WENDLAND2 : Wendland covariance (4,2)
##   27 -      MARKOV : Markovian covariances
##   28 -   GEOMETRIC : Geometric (Sphere only)
##   29 -     POISSON : Poisson (Sphere only)
##   30 -   LINEARSPH : Linear (Sphere only)
```

```
## NULL
```

Note, if you included many basic structures into the auto-fitting algorithm of gstlearn, “Model_fit” function (called below using “data.model.omni$fit” method) will try to fit every included structure and pick out the best fit to plot with the experimental variogram.

8.0.1 Omnidirectional Variogram Model

Be sure to look at the information printed out on the omnidirectional variogram.

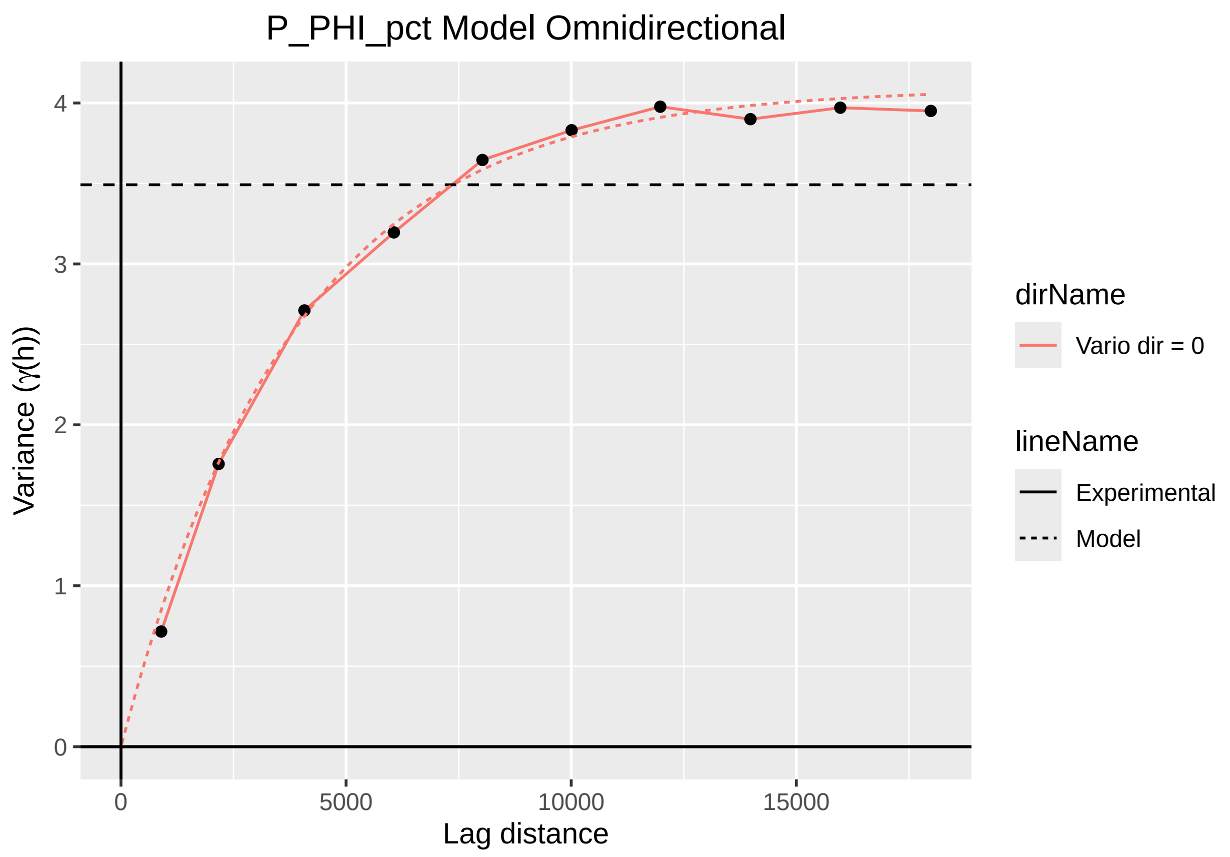
Run the chunk below…

```r
types = ECov_fromKeys(c("EXPONENTIAL"))

data.model.omni <- Model()
err = data.model.omni$fit(vario_omni, types=types)
data.model.omni
```

```
##
## Model characteristics
## =====================
## Space dimension              = 2
## Number of variable(s)        = 1
## Number of basic structure(s) = 1
## Number of drift function(s)  = 0
## Number of drift equation(s)  = 0
##
## Covariance Part
## ---------------
## Exponential
## - Sill         =      4.091
## - Range        =  11500.465
## - Theo. Range  =   3838.950
## Total Sill     =      4.091
## Known Mean(s)     0.000
```

We can more fully annotate the variograms; change the curve colors, line type and width, subtitles, enhanced legends, etc. Below is an example.

```
p = plot.init()
p = p + plot.varmod(vario_omni, data.model.omni, flagLegend = TRUE)
p = p + plot.decoration(title = paste(property, "Model Omnidirectional"), xlab = "Lag distance",
                        ylab = expression(paste("Variance (", gamma, "(h))", sep = "")))
plot.end(p)
```



```
# Step 3: Extract lag distances
gamma_exp = vario_omni$getAllGg()

# Step 4: Evaluate model values at those lags
lags = vario_omni$getAllHh()
mode = CovCalcMode()
mode$setAsVario(TRUE)
```

```
## NULL
```

```
model_vals = data.model.omni$evalIvarNlag(lags,mode=mode)

# Step 5: Compute sum of squared differences
squared_diff <- sum((gamma_exp - model_vals)^2)
print(paste0("Sum of squared difference = ", round(squared_diff,4)))
```

```
## [1] "Sum of squared difference = 0.0528"
```

9 Unique neighborhood

In the chunk below, we're going to create a unique neighborhood using function neigh.create() with type "0" for Unique Neighborhood and nidm = 2 as we are in two dimensions. Recall, a unique neighborhood uses all the data.

Run the chunk below…

```
neigh.unique <- NeighUnique_create()
```

At the same time, if you want to experiment with creating a moving neighborhood for further purposes you may. Because there's not a need for moving neighborhood in this script, it's best to comment it out to decrease the running time of the program.

Moving Neighborhood The inactive code below can be used to construct a moving neighborhood

```
#neigh.moving <- NeighMoving_create(nmaxi=10, radius=10, nsect=8, nsmax=2)
```

10 Kriging

Kriging is the geostatistical interpolation method that uses the variogram to weight neighboring data by distance and azimuth. (azimuth is an angular measurement in a spherical coordinate system).

The different about Kriging comes from the fact that the weights come from the spatial (variogram) model using the following equation: Z(0) = SUM(i=1 to n)(lambda(i)*Z(i))

10.1 Creating the Grid for Kriging

Run the chunk below…

```
dbgrid2 <- DbGrid_createCoveringDb(db_noout.db,dx=c(100,100))
dbgrid2
```

```
##
## Data Base Grid Characteristics
## ==============================
##
## Data Base Summary
## -----------------
## File is organized as a regular grid
## Space dimension              = 2
## Number of Columns            = 2
## Total number of samples      = 65844
##
## Grid characteristics:
## ---------------------
## Origin :   1198.000   670.000
## Mesh   :    100.000   100.000
## Number :        177       372
##
## Variables
## ---------
## Column = 0 - Name = x1 - Locator = x1
## Column = 1 - Name = x2 - Locator = x2
```

10.2 Kriging the variable of interest

The chunk of code below performs kriging of the selected property on a grid. It takes about a minute to run. Because of the small number of data, the kriging in this script will be performed in Unique Neighborhood. The use of the **radix** enables the creation of a second set of variables avoiding the confusion with those that already exist. Note below, radix = "Omni" creates a set of variables with the prefix, "Omni.P_PHI_pct." Look at the results of dbgrid3 once the code is run. You can see the two variables it created; Omni.P_PHI_pct.estim and Omni.P_PHI_pct.stdev. Now, all we need to do is map them.

Run the chunk below…

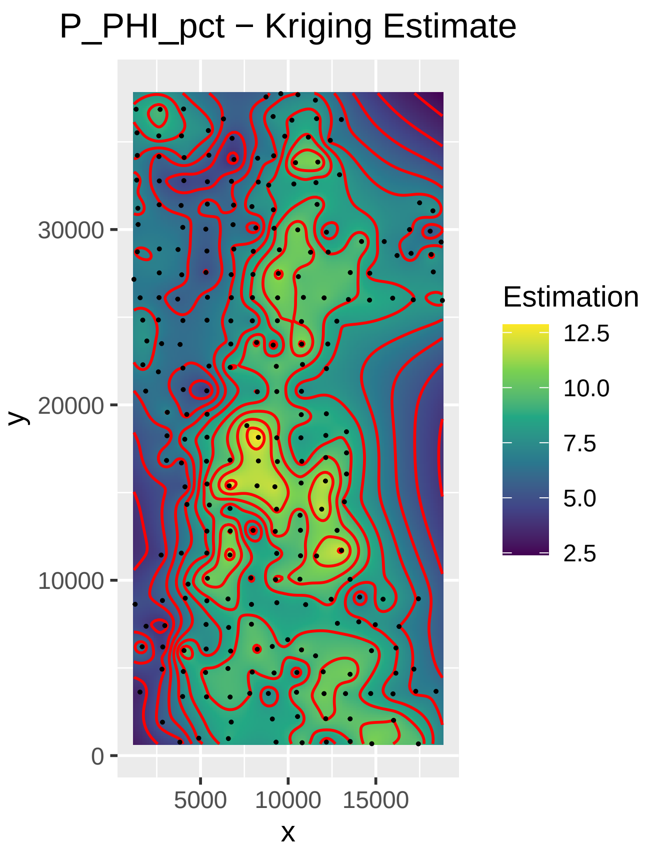
```r
err <- kriging(
    dbin = db_noout.db,
    dbout = dbgrid2,
    model = data.model.omni,
    neigh = neigh.unique,
    namconv = NamingConvention("Omni"))
dbgrid2
```

```
##
## Data Base Grid Characteristics
## ==============================
##
## Data Base Summary
## -----------------
## File is organized as a regular grid
## Space dimension              = 2
## Number of Columns            = 4
## Total number of samples      = 65844
##
## Grid characteristics:
## ---------------------
## Origin :    1198.000    670.000
## Mesh   :     100.000    100.000
## Number :         177        372
##
## Variables
## ---------
## Column = 0 - Name = x1 - Locator = x1
## Column = 1 - Name = x2 - Locator = x2
## Column = 2 - Name = Omni.P_PHI_pct.estim - Locator = z1
## Column = 3 - Name = Omni.P_PHI_pct.stdev - Locator = NA
```

From that we can plot the kriged map

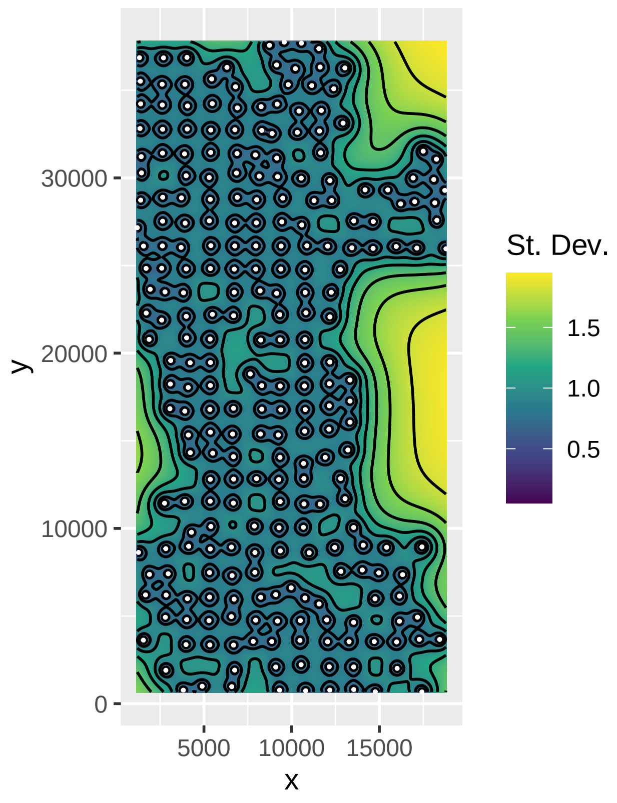
Run the chunk below…

```r
p = plot.init(asp=1)
p = p + plot.raster(dbgrid2, name = "*.estim", flagLegend=TRUE, legendName="Estimation")
p = p + plot.contour(dbgrid2, name = "*.estim", color = 'red')
p = p + plot.symbol(db_noout.db, nameSize = property, pch = 19, cex = 0.2)
p = p + plot.decoration(title = paste(property, "- Kriging Estimate"))
plot.end(p)
```



```
p = plot.init(asp=1)
p = p + plot.raster(dbgrid2, name = "*.stdev", flagLegend=TRUE, legendName = "St. Dev.")
p = p + plot.contour(dbgrid2, name = "*.stdev", color = 'black')
p = p + plot.symbol(db_noout.db, nameSize = property, pch = 19, cex = 0.2, color= 'white')
p = p + plot.decoration(title = paste(property, "- Kriging Error St. Deviation"))
plot.end(p)
```

P_PHI_pct − Kriging Error St. Deviation


y
x
St. Dev.