Contract No:

This document was prepared in conjunction with work accomplished under Contract No. DE-AC09-08SR22470 with the U.S. Department of Energy (DOE) Office of Environmental Management (EM).

Disclaimer:

A Triangulation Method for Identifying Hydrostratigraphic Locations of Well Screens

T. S. Whiteside

January 2015

SRNL-STI-2015-00020, Revision 0

DISCLAIMER

Printed in the United States of America

Prepared for
U.S. Department of Energy

SRNL-STI-2015-00020
Revision 0

Keywords: *Hydrostratigraphic, triangulation*

Retention: *Permanent*

A Triangulation Method for Identifying Hydrostratigraphic Locations of Well Screens

T. S. Whiteside

January 2015

Prepared for the U.S. Department of Energy under contract number DE-AC09-08SR22470.

Savannah River National Laboratory™
OPERATED BY SAVANNAH RIVER NUCLEAR SOLUTIONS

REVIEWS AND APPROVALS

AUTHORS:

_____

T. S. Whiteside, Radiological Performance Assessment Date

TECHNICAL REVIEW:

_____

G. A. Taylor, Radiological Performance Assessment Date

APPROVAL:

_____

M. A. Phifer Date
Radiological Performance Assessment

_____

D. A. Crowley, Manager Date
Radiological Performance Assessment

_____

K. M. Kostelnik, Manager Date
Environmental Restoration Technologies

EXECUTIVE SUMMARY

A method to identify the hydrostratigraphic location of well screens was developed using triangulation with known locations. This method was applied to all of the monitor wells being used to develop the new GSA groundwater model. Results from this method are closely aligned with those from an alternate method which uses a mesh surface.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GAU	Gordon Aquifer Unit
GCU	Gordon Confining Unit
GSA	General Separations Area
LAZ	Lower Aquifer Zone (of Upper Three Runs Aquifer)
MBCS	Meyers Branch Confining System
SRNL	Savannah River National Laboratory
TCCZ	Tan Clay Confining Zone
TZ	Transmissive Zone
UAZ	Upper Aquifer Zone
UTM	Universal Transverse Mercator

1.0 Introduction

In support of a new General Separations Area (GSA) groundwater model, this work was performed to methodically identify the hydrostratigraphic layers in which monitor wells (used in the new model) are screened.

SRS stratigraphic layers and associated nomenclature are described in a previous model (WSRC-TR-96-0399), which incorporates data from geologic cores, cone penetrometer tests, monitor wells, and other information (Figure 6-1) to define a hydrogeological framework for the GSA. The hydrostratigraphic layers used in this work are abbreviated as shown in Table 6-1. Locations for the dataset are shown in Figure 6-2 (teal colored points within the teal rectangle).

The monitor wells assessed are located within the region delineated by the purple rectangle of Figure 6-2. Each well has three points of interest: 1) the well screen top, 2) the well screen bottom, and 3) the bottom of the well (defined as 5 feet below the well screen bottom). This work describes a method to locate these points with respect to the defined hydrostratigraphic layers.

2.0 Experimental Procedure

The hydrostratigraphic layer for each screen top, screen bottom, and well bottom was determined using the following method. For each hydrostratigraphic layer (working from the Upper Aquifer Zone down to the Meyers Branch Confining System), the three nearest points of that layer that triangulate the well point, in the x-y plane, were found. These three points define a plane and the z-coordinate of the well point, relative to this plane, determines if the point is located either above, below, or in the plane. If above the plane, the well point is in the overlying hydrostratigraphic layer. If in the plane, the well point is in the hydrostratigraphic layer. If below the plane, the well point is either in the hydrostratigraphic layer being examined, or in an underlying hydrostratigraphic layer.

As shown in Figure 6-2, the monitor wells (located within the purple rectangle) are surrounded by the surface penetrations (teal colored points within the teal rectangle) used to define the hydrostratigraphic layers. A visualization of the points that define the hydrostratigraphic layers, as identified in the WSRC-TR-96-0399, is shown in Figure 6-3. The Upper Aquifer Zone (UAZ) is not depicted because the UAZ extends from the ground surface to the top of the TCCZ. The figure shows how close the TCCZ and LAZ layers are to each other in some places, as well as the closeness of the GCU and GAU layers.

For this exercise, each monitor well has three well points of interest -- well screen top, well screen bottom, and bottom of well -- and each of these points is spatially defined by UTM coordinates and by the elevation above sea level. To determine the hydrostratigraphic layer, each well point was checked to see if it could be surrounded (by latitude and longitude) by three layer points. This bounding was accomplished by finding the distance between the well point and each layer point. These distances were sorted from closest to furthest and sets of three points were analyzed, starting with the closest three, to see if they bounded the well point. First, the vectors from the well point (P) to layer points A, B, and C (PA, PB, PC) were defined. The vector describes the distance and direction from P to the point. Next, the vectors from P to the negative of point A and point B (PA′, PB′) were defined, where if the vector from P to A is (2, 2), the vector from P to A′ is (-2,-2). Then the signs of the cross product of PA′ and PB′, PA′ and PC, and PC and PB′ were determined (note the order of operations). If all three signs are equal, then the point C lies between points A’ and B’, which means points A, B, and C triangulate (bound) point P. If the signs are not all equal, points A, B, and C do not triangulate (bound) point P.

The hydrostratigraphic layer elevation is not constant; therefore, the smallest triangle that surrounds point P will provide the most accurate information about the well point. If the closest three points triangulate P,

those are the points used to create the plane. If the closest three do not triangulate P, each combination of three points within larger and larger sets have to be examined: ABCD, ABCDE, ABCDEF, …, until a triangulation is found.

Once a triangle is found within a combination, the "size" of this triangle is calculated by summing the magnitude of the vectors from P to the points in the triangle. The smallest triangle defines the best set of points surrounding a monitor well. To find the smallest triangle, all of the combinations of layer points which have a distance from point P less than or equal to the "size" of the first triangle found are analyzed to find those that triangulate P. Then these sets of points are sorted by triangle "size" and the set of points that create the smallest of these are used to define the plane. This algorithm implements finding the geometric median or "minimum sum of distances to a point" as described by Gareth Rees at the webpage listed in the Reference section.

If the well point could be triangulated by the layer points, the normal vector from the plane formed by the three points that triangulate P (ABC) and P was found. The point on the plane where this normal vector originates was found. If the well point elevation was less than or equal to the plane point's elevation, then the well point is below or in the layer. If it is greater, then it is in the overlying layer.

If the well point could not be triangulated by points in the layer, the mean elevation of the layer was compared to the well point elevation. The mean and median elevations for each layer were examined and found to be similar, so we chose to use the mean elevation and treat the layer as having no slope, as the layers are mostly level, see Figure 6-3, and this is simpler to compute. If the well point elevation was below or equal to the mean elevation, the well point is below or in the hydrostratigraphic layer. If the well point's elevation was above the mean elevation, the well point is in the overlying layer.

3.0 Results and Discussion

The dataset for this work is from 1052 groundwater monitor wells and includes UTM coordinate data and the elevations of the well screen top, well screen bottom, and the bottom of the well (the well screen bottom elevation minus 5 ft). Each monitor well has three well points of interest -- well screen top, well screen bottom, and bottom of well -- and each of these points is spatially defined by UTM coordinates and by the elevation above sea level. A Python program was written to determine the hydrostratigraphic layer of each well point, implementing the above method. This program's output identified the hydrostratigraphic unit of each well point and if the components of the well screen crossed hydrostratigraphic layers.

In this program each well point was examined to see if it could be triangulated by the layer above and the layer below. If the well point could be triangulated by both surfaces, the abbreviation of the hydrostratigraphic unit was output. If the well point could not be triangulated by the previous layer a "???" was placed in front of the output; if it could not be triangulated by the current layer a "???" was placed behind the output. See Table 6-2 for a sample of the program output. If the hydrostratigraphic units for all three well points are not all the same, then the well screen crosses hydrostratigraphic layers and the STATION_NAME field is proceeded by a "***".

The hydrostratigraphic units are not as cleanly defined as depicted in Figure 6-3. Figure 6-4 plots the elevations vs UTM Eastings of the Lower Aquifer Zone for the area of interest. This plot shows the variability in the hydrostratigraphic surface and the median elevation (180 ft). This surface variability is likely the cause of some of the cross-unit identifications in the triangulation method.

Overall, the method described here is a good method to quickly identify a majority of hydrostratigraphic units for a selection of points. However, because the triangle selection algorithm scales as $O(n^3)$, if a

triangle is not found within the first few points it takes much longer to solve. In the collection of wells provided, twelve wells (36 points) required nearly three days of computation time to complete. All of the remaining 3120 points were located in less than 30 minutes.

To validate this method, the results were compared to those determined by locating the parts of the well using a mesh constructed from the original GSA model coordinates. For the well series BG26-BG67, only one well point assigned to a different adjacent layer (out of 105). For the well series BGO1D-BGO53D, there were 20 points assigned to different adjacent layers (out of 414 well points). These differences in assignment are most likely due to how the triangles were constructed.

4.0 Conclusions

A collection of hydrostratigraphic unit data was used to identify the unit(s) in which various groundwater modeling wells are located. While straightforward to describe and implement, the algorithm does not account for sparse data (large triangles) or locations where there is wide variability between elevations of the nearest points. If this tool is used in future work, these issues should be addressed as well as further optimizations of the algorithm.

5.0 Reference

Smits, A. D.; Harris, M. K.; Hawkings, K. L.; Flach, G. P. Integrated Hydrogeological Modeling of the General Separations Area Volume 1: Hydrogeologic Framework. Aug, 1997. WSRC-TR-96-0399 Rev. 0.

Sum of distances algorithm website: http://stackoverflow.com/questions/4229454/algorithm-to-find-the-closest-3-points-that-when-triangulated-cover-another-poin

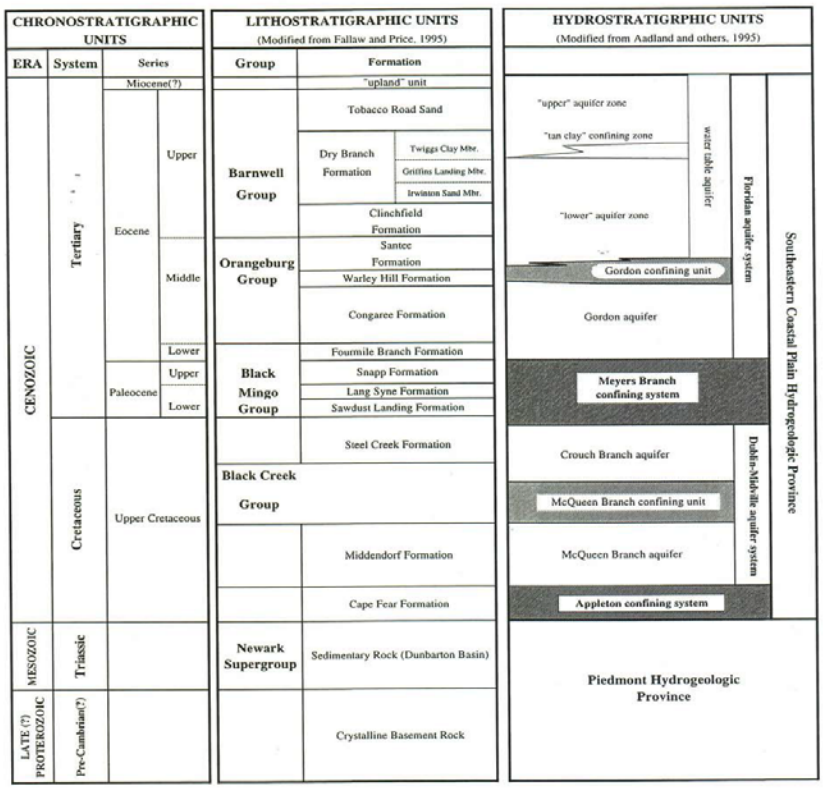

Figure 5-1. Regional Stratigraphic Nomenclature (ref), including Hydrostratigraphic Layers used in this work.

Table 5-1. Table 1. Table of Hydrostratigraphic Layers used in the model

Hydrostratigraphic layer	Abbreviation
Upper Aquifer Zone	UAZ
Tan Clay Confining Zone	TCCZ
Lower Aquifer Zone (of Upper Three Runs Aquifer)	LAZ
Gordon Confining Unit	GCU
Gordon Aquifer Unit	GAU
Meyers Branch Confining System	MBCS

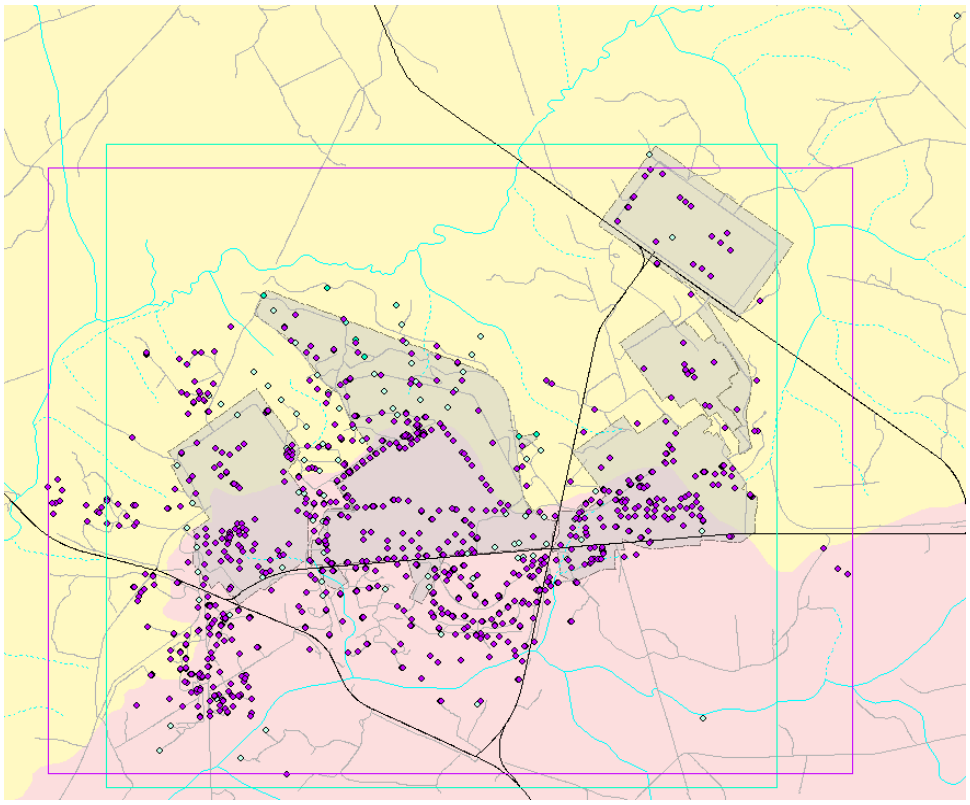

Figure 5-2. Teal rectangle bounds the stratigraphic data used in WSRC-TR-96-0399. Purple points delineate the monitor wells dataset.

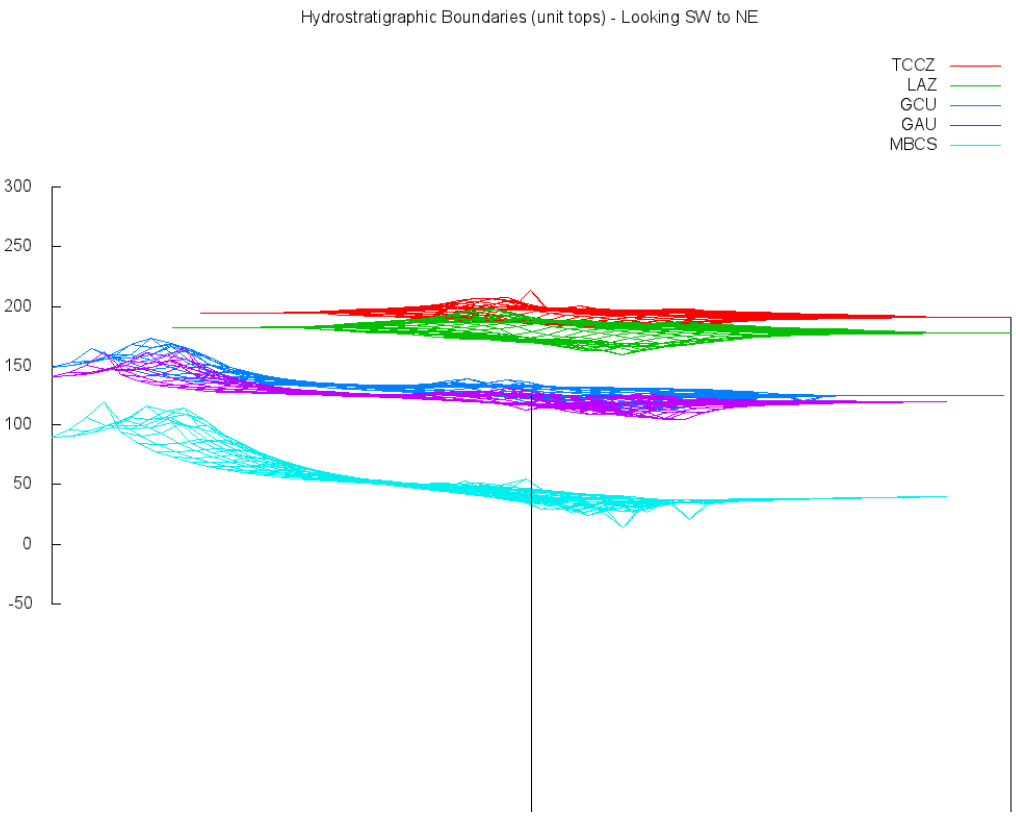

Figure 5-3. Elevation of hydrostratigraphic layers at SRS

Table 5-2. First thirty rows of output of Analysis Program

STATION_NAME	UTM_EAST	UTM_NORTH	SZ Top El.	HUNIT	SZ Bot El.	HUNIT	Bot-5ft	HUNIT
BG 26	438952.14	3682915.89	230.7	#UAZ	210.7	#UAZ	205.7	#UAZ
BG 27	438880.79	3683014.07	254.4	#UAZ	234.4	#UAZ	229.4	#UAZ
BG 28	438809.95	3683111.49	259.7	#UAZ	239.7	#UAZ	234.7	#UAZ
BG 29	438738.21	3683209.88	251.6	#UAZ	231.6	#UAZ	226.6	#UAZ
BG 30	438666.55	3683307.9	251.7	#UAZ	231.7	#UAZ	226.7	#UAZ
BG 31	438593.53	3683405.43	243.3	#UAZ	223.3	#UAZ	218.3	#UAZ
BG 32	438521.74	3683503.93	246.9	#UAZ	226.9	#UAZ	221.9	#UAZ
BG 33	438430.07	3683486.19	241.2	#UAZ	221.2	#UAZ	216.2	#UAZ
***BG 34	438324.49	3683414.5	237.4	#UAZ	217.4	#UAZ	212.4	#TCCZ???
BG 35	438230.32	3683346.59	248.0	#UAZ	228.0	#UAZ	223.0	#UAZ
***BG 36	438158.94	3683389.72	243.3	#UAZ	223.3	#UAZ	218.3	#TCCZ???
BG 37	438057.69	3683337.61	247.8	#UAZ	227.8	#UAZ	222.8	#UAZ
BG 38	437959.16	3683265.92	245.9	#UAZ	225.9	#UAZ	220.9	#UAZ
BG 39	437860.68	3683194.2	246.0	#UAZ	226.0	#UAZ	221.0	#UAZ
BG 40	437762.03	3683122.46	241.9	#UAZ	221.9	#UAZ	216.9	#UAZ
BG 41	437758.18	3683033.43	241.0	#UAZ	221.0	#UAZ	216.0	#UAZ
***BG 42	437829.63	3682935.63	237.1	#UAZ	217.1	#UAZ	212.1	#TCCZ???
BG 43	437930.0	3682885.71	242.9	#UAZ	222.9	#UAZ	217.9	#UAZ
BG 51	438917.18	3682854.98	241.2	#UAZ	221.2	#UAZ	216.2	#UAZ
BG 52	437792.72	3682807.6	243.8	#UAZ	223.8	#UAZ	218.8	#UAZ
***BG 53	437637.49	3682787.65	234.7	#UAZ	214.7	#TCCZ???	209.7	#TCCZ???
BG 54	437634.76	3682665.29	235.2	#UAZ	215.2	#UAZ	210.2	#UAZ
BG 55	437631.74	3682545.27	234.9	#UAZ	214.9	#UAZ	209.9	#UAZ
BG 56	437662.16	3682447.26	230.9	#UAZ	210.9	#UAZ	205.9	#UAZ
BG 57	437782.41	3682457.12	234.6	#UAZ	214.6	#UAZ	209.6	#UAZ
BG 58	437904.31	3682466.89	238.2	#UAZ	218.2	#UAZ	213.2	#UAZ
BG 59	438024.96	3682480.29	237.7	#UAZ	217.7	#UAZ	212.7	#UAZ
BG 60	438146.35	3682490.6	235.5	#UAZ	215.5	#UAZ	210.5	#UAZ
BG 61	438327.87	3682505.56	245.0	#UAZ	225.0	#UAZ	220.0	#UAZ
BG 62	438388.39	3682510.49	242.5	#UAZ	222.5	#UAZ	217.5	#UAZ

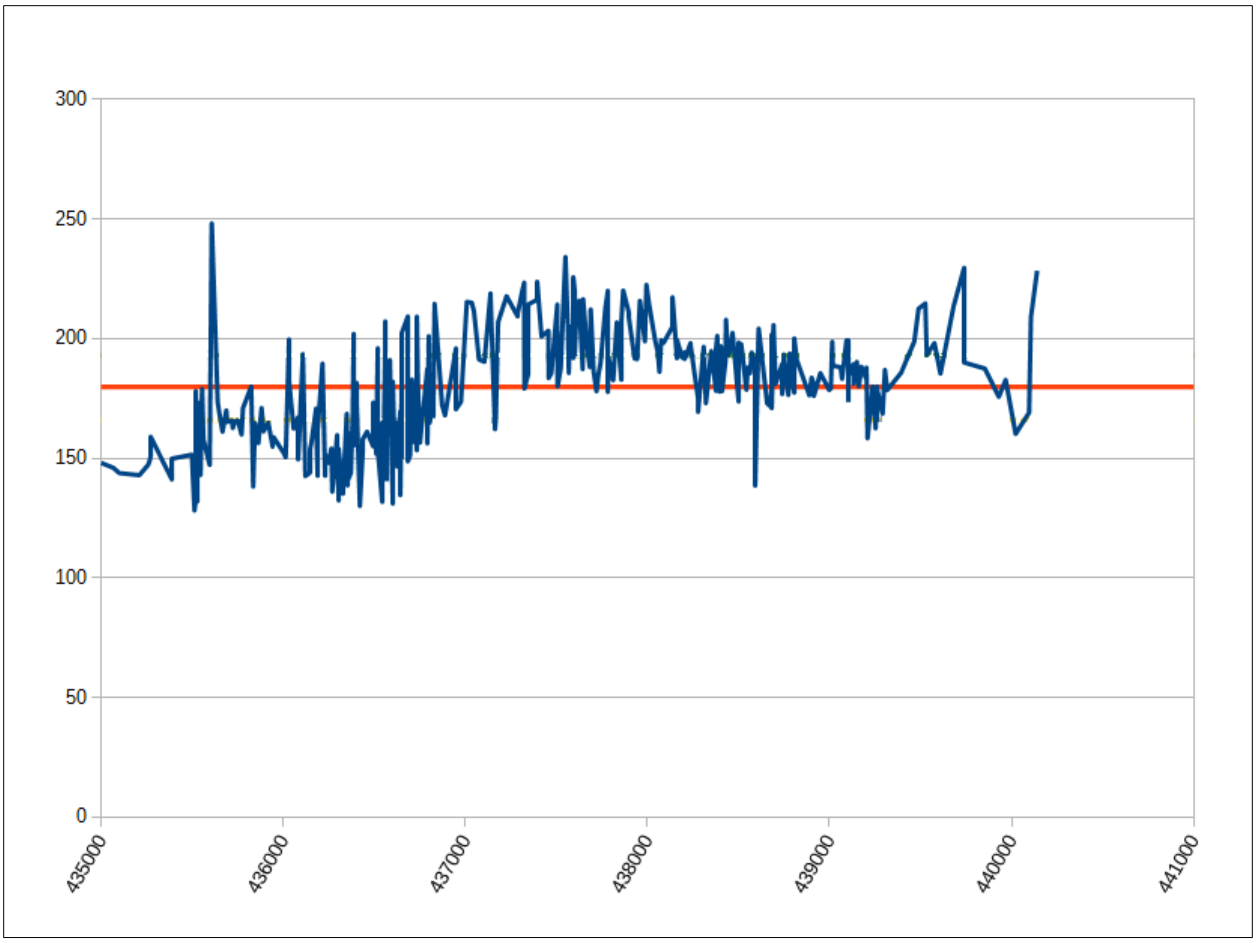

Figure 5-4. UTM easting vs. elevation of the LAZ, showing the median elevation

Appendix A.

Appendix A. Program Listing

```
#!/hpc/apps/python-2.7-srnl/bin/python

import itertools
import math
import numpy
import collections
import sys
import multiprocessing
from scipy.spatial.distance import pdist, euclidean

def get_vector(A, B):
    U = numpy.subtract(B, A)

#    U = B - A
#    U = U.sub(A,B)
#    Ux = B.x - A.x
#    Uy = B.y - A.y
#    Uz = B.z - A.z

#    U = Point(Ux, Uy, Uz)
    return U
#enddef

def normal_point_plane_intersection(P, A, B, C):
    #find equation of plane [A, B, C]
    #ax + by + cz + d = 0

    #two vectors to define the plane
    U = get_vector(A, B)
    V = get_vector(A, C)

    #cross product of UxV
    UxV = numpy.cross(U, V)
    a = UxV[0]
    b = UxV[1]
    c = UxV[2]

#    a = U.y*V.z - U.z*V.y
#    b = -1.0 * (U.x*V.z - U.z*V.x)
#    c = U.x*V.y - U.y*V.x
#    d = -1.0*(a*A.x + b*A.y + c*A.z)

    d = -1.0*(a*A[0] + b*A[1] + c*A[2])

    #normal vector from plane
    #N = Point(a, b, c)
    N = numpy.array([a, b, c])

    #find the point on the plane that the normal vector from P passes through
    #R = (P.x,P.y,P.z) + tau(-N.x, -N.y, -N.z)
    #x = P.x + tau*-N.x, y = P.y + tau*-N.y, z = P.z + tau*-N.z
    #sub x,y,z into ax + by + cz + d = 0
    #solve for tau, which is:

#    tau = (d + a*P.x + b*P.y + c*P.z)/(a*N.x + b*N.y + c*N.z)
    tau = (d + a*P[0] + b*P[1] + c*P[2])/(a*N[0] + b*N[1] + c*N[2])

    ix = (P[0] + tau * -1.0 * N[0])
    iy = (P[1] + tau * -1.0 * N[1])
    iz = (P[2] + tau * -1.0 * N[2])

#    I = Point(ix, iy, iz)
    I = numpy.array([ix, iy, iz])

#    print a, b, c, d
#    equation of plane: z = d + mx + ny
```

```
#    print str(-d/c) + "+" + str(-a/c) + "x+" + str(-b/c)+ "y"
    """
    dat = ""
    if c <> 0.0:
        dat = dat + "#" + str(-d/c) + "+" + str(-a/c) + "x+" + str(-b/c)+ "y" + "\n"
    else:
        print "come review well at :" + str(P.x) + " " + str(P.y) + " " + str(P.z)
    #endif
    dat = dat + str(A.x)+ " " + str(A.y) + " " + str(A.z) + "\n"
    dat = dat + str( B.x) + " " + str( B.y) + " " + str( B.z) + "\n"
    dat = dat + str( C.x) + " " + str( C.y) + " " + str( C.z) + "\n"
    dat = dat + "\n"
    dat = dat + "\n"
    dat = dat + str( P.x) + " " + str( P.y) + " " + str( P.z) + "\n"
    dat = dat + "\n"
    dat = dat + "\n"
    dat = dat + str( A.x) + " " + str( A.y) + " " + str( A.z) + " " + str( N.x/N.x) + " " + str(
N.y/N.x) + " " + str( N.z/N.x) + "\n"
    dat = dat + str( B.x) + " " + str( B.y) + " " + str( B.z) + " " + str( N.x/N.x) + " " + str(
N.y/N.x) + " " + str( N.z/N.x) + "\n"
    dat = dat + str( C.x) + " " + str( C.y) + " " + str( C.z) + " " + str( N.x/N.x) + " " + str(
N.y/N.x) + " " + str( N.z/N.x) + "\n"
    dat = dat + "\n"
    dat = dat + "\n"
    dat = dat + str( P.x) + " " + str( P.y) + " " + str( P.z) + " " + str( -N.x/N.x) + " " + str(
-N.y/N.x) + " " + str( -N.z/N.x) + "\n"
    dat = dat + "\n"
    dat = dat + "\n"
    dat = dat + str( P.x) + " " + str( P.y) + " " + str( P.z) + "\n"
    dat = dat + str( I.x) + " " + str( I.y) + " " + str( I.z) + "\n"
    dat = dat + str( P.x +( 2*tau*-N.x)) + " " + str(P.y +( 2*tau*-N.y)) + " " + str(P.z +(
2*tau*-N.z)) + "\n"
    """
#    f = open("pp.txt", "w")
#    f.write(dat)
#    f.close()

    return I
#enddef

def point_point_distance(p, q):
    distance = numpy.linalg.norm(p - q)
    return distance
#enddef

def get_max_min_point(max_min, array, index):
    #get the index value of the array slice (column of array) with max/min value
    if max_min == 'max':
        i_val = array[:,index].argmax()
    else:
        i_val = array[:,index].argmin()
    #endif
    #get the point using the i_val (and then cut off the distance part)
    point = array[i_val]
    point = point[:3]
    return point
#enddef

def get_lnn2_aux(input_list):
    P = input_list[0]
    apicks = input_list[1]
    lnn        = []
    hdistances = {}
    for pick in apicks:
        distance = numpy.sqrt(numpy.sum((P-pick)**2))
#        distance = numpy.linalg.norm(P-pick)
#        distance = euclidean(P, pick)
        lnn.extend([numpy.append(pick,distance)])
        hdistances[str(pick[0])+str(pick[1])+str(pick[2])] = distance
    #endfor
    return lnn, hdistances
```

```
#enddef

def get_lnn2(P, apicks):
    eastings   = []
    northings  = []
    elevations = []

    """
    a = numpy.arange(12).reshape(3,4)
    print a
    b = pdist(a, 'euclidean')
    print b

    b = euclidean(P,apicks[0])
    c = numpy.linalg.norm(P-apicks[0])
    d = numpy.sqrt(numpy.sum((P-apicks[0])**2))
    print b, c, d

    exit()
    """

#    lnn, hdistances = get_lnn2_aux(P, apicks)
#    pool = multiprocessing.Pool()
#    lnn, hdistances = pool.map(get_lnn2_aux, [P, apicks])
#    pool.close
#    pool.join

    lnn, hdistances = get_lnn2_aux([P, apicks])

    lnn_array = numpy.array(lnn)
    max_easting_point   = get_max_min_point('max', lnn_array, 0)
    min_easting_point   = get_max_min_point('min', lnn_array, 0)
    max_northing_point  = get_max_min_point('max', lnn_array, 1)
    min_northing_point  = get_max_min_point('min', lnn_array, 1)
    max_elevation_point = get_max_min_point('max', lnn_array, 2)
    min_elevation_point = get_max_min_point('min', lnn_array, 2)

#    mean_elevation  = numpy.average(lnn_array[:,2])
#    mean_elevation  = numpy.mean(lnn_array[:,2])
    mean_elevation  = numpy.median(lnn_array[:,2])

    lnn_array = lnn_array[lnn_array[:,3].argsort()]
    lnn = numpy.delete(lnn_array,[3],axis=1) #.tolist()

    return lnn, hdistances, max_easting_point, min_easting_point, max_northing_point,
min_northing_point, max_elevation_point, min_elevation_point, mean_elevation
#enddef


def get_hdistances(lnn):
    hdistances = {}
    for p in lnn:
        hdistances[str(p[0])+str(p[1])+str(p[2])] = p[3]
    #endfor

    return hdistances
#enddef


def xy_angle(U, V):
    #U.V = |U|*|V| * cos(theta)
    #UxV = |U|*|V| * sin(theta) * n

    UdotV = U.x * V.x + U.y * V.y
    magU  = math.sqrt(U.x**2 + U.y**2)
    magV  = math.sqrt(V.x**2 + V.y**2)

    UxV =  U.x*V.y - U.y*V.x

    theta = math.acos(UdotV/(magU *magV))
```

```
    n = UxV/(magU * magV * math.sin(theta))
    if n < 0:
        theta = theta + math.pi
    #endif

    theta = theta * 180/math.pi

    return theta
#enddef

def plot_2d(P, A, B, C):
    print P.x, P.y
    print ""
    print ""
    print P.x, P.y, A.x-P.x, A.y-P.y
    print P.x, P.y, B.x-P.x, B.y-P.y
    print P.x, P.y, C.x-P.x, C.y-P.y
    print ""
    print ""
    print "A", A.x, A.y
    print "B", B.x, B.y
    print "C", C.x, C.y

#enddef

def sign_cross_product(U, V):
    UxV = numpy.cross(U, V)

#    UxV = U.x*V.y - U.y*V.x
#    print UxV
    sign = 0
    if UxV[2] > 0:
        sign = 1
    elif UxV[2] < 0:
        sign = -1
    else:
        sign = 0
    #endif
    return sign
#enddef


#find if P can be triangulated by lnn by:
#    look at first 3 points
#    if not found, look at combinations of first 4, then 5, then 6, etc
#    once a triangle is found, you are limited to the lnn that the distance from P is equal to
the sum of distances of the first triangle found.
#    get all of these points (lnn[0,...,distance <= sum_distance_first_triangle])
#    check all these combinations, the best triangle is the one with the minimal sum of distances

def find_abc(lnn, hdistances, P):

    for i in range(2,len(lnn)): #the first two are used as A, B, keep extending the possible C
        num = 0
        c_ok = False
        possible_points = lnn[:i+1]

        for combo in itertools.combinations(possible_points, 3):
            A = combo[0]
            B = combo[1]
            C = combo[2]

            #print "======="
            #print A
            #print B
            #print C


            sAoC, sCBo, sAoBo = find_abc_aux2(A, B, C, P)

            #print sAoC, sCBo, sAoBo
```

```
            if sAoC == sCBo and sCBo == sAoBo :        #C is good
                lpp = len(possible_points)
                if lpp == 3:
                    #the first 3 points cover the Point, so get out
                    #print "C is good"
                    c_ok = True
                    num = 3
                    break
                else:
                    #found a point that works - now need to find the best point

                    hindexA = get_hdistance_index(A)
                    hindexB = get_hdistance_index(B)
                    hindexC = get_hdistance_index(C)

                    sum_dist = hdistances[hindexA] + hdistances[hindexB] + hdistances[hindexC]

                    print "Sum:" + str(sum_dist)

                    j = 0
                    k = 0
                    for q in lnn:
                        hindexQ = get_hdistance_index(q)
                        dist = hdistances[hindexQ]
                        if dist <= sum_dist:
                            #print j, dist
                            k = j
                        #endif
                        j= j+1
                    #endfor

                    #need to optimize
                    possible_points = lnn[:k+1]

#                    print len(possible_points)
#                    print possible_points

#                    if sum_dist > 1000:
#                        print "Too many possiblilties, quitting for now"
#                        c_ok = False
#                        break
#                    #endif


                    A, B, C = find_abc_aux3(possible_points, hdistances, P)
                    num = 3
                    c_ok = True
                    break
                #endif
            #endif
        #endfor
        if num == 3:
            break
        #endif
    #endfor

    lABC = [c_ok, A, B, C]
    return lABC
#enddef

def get_hdistance_index(A):

    index = str(A[0]) + str(A[1]) + str(A[2])

    return index
#enddef


def find_abc_aux(lnn, hdistances, P):
    lcombos = []
```

```
    #combos = 0
    for combo in itertools.combinations(lnn, 3):
        A = combo[0]
        B = combo[1]
        C = combo[2]

        sAoC, sCBo, sAoBo = find_abc_aux2(A, B, C, P)

        if sAoC == sCBo and sCBo == sAoBo :        #C is good
            #calculate sum of distances of these points
            hindexA = get_hdistance_index(A)
            hindexB = get_hdistance_index(B)
            hindexC = get_hdistance_index(C)

            sum_dist = hdistances[hindexA] + hdistances[hindexB] + hdistances[hindexC]

            lcombos.append([sum_dist,[A,B,C]])
        #endif
        #combos = combos + 1
    #endfor

    #print combos

    #sort the combo list by sum_distances, return the shortest
    lshort = sorted(lcombos, key=lambda combo: combo[0])

    #print lshort[0]
    #print lshort[0][0]
    #print lshort[0][1][0].x
    #print lshort[0][1][1].x
    #print lshort[0][1][2].x

    A = lshort[0][1][0]
    B = lshort[0][1][1]
    C = lshort[0][1][2]

    return A, B, C
#enddef

def find_abc_aux3(lnn, hdistances, P):
    lcombos = []
    for combo in itertools.combinations(lnn, 3):
        A = combo[0]
        B = combo[1]
        C = combo[2]

        #calculate sum of distances of these points
        hindexA = get_hdistance_index(A)
        hindexB = get_hdistance_index(B)
        hindexC = get_hdistance_index(C)

        sum_dist = hdistances[hindexA] + hdistances[hindexB] + hdistances[hindexC]

        lcombos.append([sum_dist,[A,B,C]])
    #endfor

    #sort the combo list by sum_distances, return the shortest
    lshort = sorted(lcombos, key=lambda combo: combo[0])

    lshort_good = []
    for combo in lshort:
        A = combo[1][0]
        B = combo[1][1]
        C = combo[1][2]

        sAoC, sCBo, sAoBo = find_abc_aux2(A, B, C, P)

        if sAoC == sCBo and sCBo == sAoBo :        #C is good
            lshort_good.extend([A,B,C])
        #endif
        if len(lshort_good) == 3:
```

```
            break
        #endif
    #endfor

    A = lshort_good[0]
    B = lshort_good[1]
    C = lshort_good[2]

    return A, B, C
#enddef


#get the sign of the cross product of vectors
def find_abc_aux2(A, B, C, P):
    PA = get_vector(P, A)
    PB = get_vector(P, B)
    PAo = -1 * PA
    PBo = -1 * PB

    #work backwards to get the opposite point
    Ao = get_vector(PA, P)
    Bo = get_vector(PB, P)

    PC = get_vector(P, C)

    sAoC  = sign_cross_product(PAo, PC)
    sCBo  = sign_cross_product(PC, PBo)
    sAoBo = sign_cross_product(PAo, PBo)

    return sAoC, sCBo, sAoBo
#enddef


#get the hydrozone (and some other info) for a point
def find_hydrozone_simpler(P, hydrodata):

    elevations = collections.OrderedDict()
    #loop over each hydrozone, see if Point is above, within, or below the layer
    #start with the '#UAZ'
    print "P can be triangulated by the #UAZ boundaries" #default

    hydrozone          = '#UAZ'
    elevations[hydrozone] = "ground surface"
    triangle_current   = True

    previous_hydrozone = '#UAZ'
    triangle_previous  = True



    for hydrozone in hydrodata.keys():
        if hydrozone in ('#AA', '#TZ'): #skip these for the GSA model
            hydrozone = previous_hydrozone
        elif hydrozone in ('#CBAU'):    #skip this one for the GSA model
            hydrozone = previous_hydrozone
        else:
            # checking to see if P can be triangulated by the " + hydrozone + " boundaries...."

            apicks = hydrodata[hydrozone]

            #get the list of nearest neighboring points (closest to furthest) in this layer
            #    and the points of the layer limits
            lnn, hdistances, max_easting, min_easting, max_northing, min_northing, max_elevation,
min_elevation, mean_elevation = get_lnn2(P, apicks)

#            print lnn


            close_val = 5.0
            diff = math.sqrt(((P[0] - lnn[0][0])**2 + (P[1] - lnn[0][1])**2)/2)
```

```
            if diff < close_val :
#            if abs(P[0] - lnn[0][0]) < close_val and abs(P[1] - lnn[0][1]) < close_val :
                print "P's (N,E) are within " + str(close_val) + "(m) of the (N,E) of a point in
the " + hydrozone
                #P is either in this hydrozone or the next one down
                elevations[hydrozone] = lnn[0][2]
                if P[2] <= lnn[0][2]:
                    print "\tP is below or in layer " + hydrozone
                    hydrozone = hydrozone
                else:
                    print "\tP is above layer " + hydrozone
                    hydrozone = previous_hydrozone
                    break
                #endif
            else:
                #see if point can be triangulated by this layer, if yes - get the smallest
triangle that surrounds the point

                #print max_easting
                #print min_easting
                #print max_northing
                #print min_northing
                #print P

                lcABC = find_abc([max_easting, min_easting, max_northing, min_northing],
hdistances, P)

                if lcABC[0] == False:
                    print "P cannot be triangulated by "  + hydrozone + " boundaries."
                    triangle_current = False

                    #set hydrozone to current_hydrozone and flag it - fix on quit if not found in
bottom layers
                    elevations[hydrozone] = str(mean_elevation) + "(mean_elevation)"
                    line = "\tUsing the mean_elevation (" + str(mean_elevation) + "), P may be "
                    if P[2] <= mean_elevation :
                        print line + "below or in layer " + hydrozone
                        hydrozone = hydrozone
                    else :
                        print line + "above layer " + hydrozone
                        hydrozone = previous_hydrozone
                        break
                    #endif
                else:
                    print "P can be triangulated by " + hydrozone + " boundaries."
                    triangle_current = True

                    #zoom in to get a better answer

                    elevations[hydrozone] = max_elevation[2]
                    if max_elevation[2] < P[2]:
                        #P is definately in the previous layer, no need to check further down
layers
                        print "\tP in " + previous_hydrozone
                        hydrozone = previous_hydrozone
                        break
                    else:
                        print "\tP is either above or below " + hydrozone + " and can be
triangulated by this hydrozone"
                        print "\tfinding closest points that when triangulated, cover P..."

                        lcABC = find_abc(lnn, hdistances, P)

                        if lcABC[0] == False:
                            print "XXXXXXXXXXX this shouldn't happen XXXXXXXXXX"
                            exit()
                        else:
                            A = lcABC[1]
                            B = lcABC[2]
                            C = lcABC[3]
```

```
                            I = normal_point_plane_intersection(P,A,B,C)

                            #print A, B, C
                            elevations[hydrozone] = I[2]
                            if P[2] <= I[2] : #P is below or in the layer
                                print "\tP is below or in layer " + hydrozone
                                hydrozone = hydrozone
                            else : #above the plane, use the previous hydrozone and quit
                                print "\tP is above layer " + hydrozone
                                hydrozone = previous_hydrozone
                                break
                            #endif  #If normal
                        #endif      #If lcABC
                    #endif          #If max_elevation
                #endif      #if lcabc
            #endif          #If close
        #endif
        previous_hydrozone = hydrozone
        triangle_previous  = triangle_current
    #endfor

    #in the last layer...

    if hydrozone == hydrodata.keys()[-1] :
        #in the bottom layer, so there is no "current" (bottom layer)
        #make the previous layer = current layer
        triangle_previous = triangle_current

        #this was to show that the bottom could be something else (confusing to regular people)
        #if triangle_current == False:
        #    hydrozone = hydrodata.keys()[-1] + "/unknown"
        #endif

        triangle_current = True
    #endif


    flag_current = False
    if triangle_previous == triangle_current and triangle_current == True:
        flag_current = False
    else:
        flag_current = True
    #endif

    """
    if triangle_previous == False:
        print "???",
    #endif
    print hydrozone,
    if triangle_current == False:
        print "???"
    #endif
    """

    return hydrozone, flag_current, triangle_previous, triangle_current, elevations
#enddef


def get_hydrodata():
    #read the hydro file
    filename = 'hydro.txt'
    f = open(filename, "r")

    #read all the lines into an array (list)
    hydrodata = f.read().splitlines()

    f.close()

    return hydrodata
#enddef
```

```
def format_zone(flag, t_prev, t_curr, hydrozone):
    line = ""

    if flag == True:
        if t_prev == False:
            line = line + "???"
        #endif

        line = line + hydrozone

        if t_curr == False:
            line = line + "???"
        #endif
    else:
        line = hydrozone
    #endif

    line = line + "\t"

    return line
#enddef

def get_hydro_datastruct(hydrodata):
    #get a list of all the 8 layers in the hydrodata (these are: AA; TZ; TCCZ; LAZ; GCCZ/GCU;
LLAZ/GAU; MBCS/CBCU; CBAU)
    #add the UAZ layer, as the default (we will skip AA and TZ when looping)

    hydrodata_struct = collections.OrderedDict()

    alayers = [list(group) for k, group in itertools.groupby(hydrodata, lambda x: x== "") if not
k]

    for layer in alayers:
        layer_id      = layer[0]
        #layer_coord  = layer[1]  #UTM_E UTM_N elevation - not used
        layer_picks   = layer[2:]

        hydrodata_struct[layer[0]] = [numpy.array([float(x) for x in pick.split('\t')]) for pick
in layer_picks]
    #endfor

    return hydrodata_struct
#enddef



def main():

    #get the hydro stratigraphic data
    hydrodata = get_hydrodata()

    #get hydrodata into a datastructure
    hydrodata_struct = get_hydro_datastruct(hydrodata)

    #open the input and output files

    #for "parallel" use
    filename = sys.argv[1]
#    filename = 'wells.txt'


    f = open(filename, "r")

    trim_filename = filename.replace(".txt","")
    trim_filename = trim_filename.replace("s","")

    outfilename = trim_filename + '_hydrozones.txt'

    of = open(outfilename, "w")
    #of.write("STATION_NAME\tSTATION_TYPE\tUTM_EAST\tUTM_NORTH\tSZ Top El.\tHUNIT\tSZ Bot
El.\tHUNIT\t\tBot-5ft\tHUNIT\tUnits\n")
```

```
    of.write("STATION_NAME\tUTM_EAST\tUTM_NORTH\tSZ Top El.\tHUNIT\tSZ Bot El.\tHUNIT\tBot-
5ft\tHUNIT\t#UAZ\t#TCCZ\t#LAZ\t#GCU\t#GAU\t#MBCS\n")

    #read all the well data lines
    lines = f.readlines()

    i = 0
    j = 0
    k = 3e22
    for line in lines:
        if i == k: #get out after k lines (delete this when done testing - or make k really big)
            i = i - 1
            break
        #endif
        if i == 0: #the first line - skip it
            i = i
        else:
            line = line.strip()
            afields = line.split('\t')
            station_name = afields[0]
            #station_type = afields[1]
            #units       = afields[7] #hardcoding below

            print station_name

            #see if the field is bad, if so skip it
            check = False
            for field in afields:
                if len(field) < 1: #this field is bad, skip this well
                    check = False
                    break
                else:
                    check = True
                #endif
            #endfor

            if check == True:
                #well_easting  = float(afields[2])
                #well_northing = float(afields[3])
                #well_top      = float(afields[4])
                #well_bot      = float(afields[5])
                #well_bm5      = float(afields[6])

                well_easting  = float(afields[1])
                well_northing = float(afields[2])
                well_top      = float(afields[3])
                well_bot      = float(afields[4])
                well_bm5      = float(afields[5])

                #define the coordinates for top, bot, bm5 of the screen
                top = numpy.array([well_easting, well_northing, well_top])
                bot = numpy.array([well_easting, well_northing, well_bot])
                bm5 = numpy.array([well_easting, well_northing, well_bm5])


                #print the triangulated elevations of each hydrolayer along side each well


                #get the hydrozone,
                hydrozone_top, top_flag, top_prev, top_curr, elevationst =
find_hydrozone_simpler(top, hydrodata_struct)
                print "The hydrozone for top of well is: " + hydrozone_top + "\n"
                hydrozone_bot, bot_flag, bot_prev, bot_curr, elevationsb =
find_hydrozone_simpler(bot, hydrodata_struct)
                print "The hydrozone for bot of well is: " + hydrozone_bot + "\n"
                hydrozone_bm5, bm5_flag, bm5_prev, bm5_curr, elevationsb5 =
find_hydrozone_simpler(bm5, hydrodata_struct)
                print "The hydrozone for bm5 of well is: " + hydrozone_bm5 + "\n"

                #Some logic that formats the output of wells with "cross-zone" screens...
                str_pre = ""
```

```
                if hydrozone_top == hydrozone_bot == hydrozone_bm5:
                    str_pre = ""
                else:
                    str_pre = "***"
                    j = j + 1
                #endif

                #write file with top, bot, bm5 and hydrozone labels
                line = str_pre + station_name + "\t" + \
                       str(well_easting) + "\t" + \
                       str(well_northing) + "\t"

                       #station_type + "\t" + \ - goes between station_name and well_easting


                line = line + str(well_top) + "\t"
                line_zone_top = format_zone(top_flag, top_prev, top_curr, hydrozone_top)
                line = line + line_zone_top

                line = line + str(well_bot) + "\t"
                line_zone_bot = format_zone(bot_flag, bot_prev, bot_curr, hydrozone_bot)
                line = line + line_zone_bot

                line = line + str(well_bm5) + "\t"
                line_zone_bm5 = format_zone(bm5_flag, bm5_prev, bm5_curr, hydrozone_bm5)
                line = line + line_zone_bm5

                for key in elevationsb5:
                    line = line + str(elevationsb5[key]) + "\t"
                #endfor

                #line = line + "ft"

            else:
                line = station_name + ":  Bad data for well<=========================="
            #endif
#            print line

            of.write(line + "\n")
            of.flush()
        #endif
        i = i+1
    #endfor
    last_line = "There are " + str(j) + " cross zone wells, out of " + str(i) + " wells\n"
    print last_line
    of.write(last_line)

    #close the files
    of.close()  #outfile (well_hydrozones.txt)
    f.close()   #infile  (wells.txt)
#enddef


if __name__ == "__main__":
    main()
#endif
```

Distribution:

S. L. Marra, 773-A
T. B. Brown, 773-A
D. H. McGuire, 999-W
S. D. Fink, 773-A
C. C. Herman, 773-A
E. N. Hoffman, 999-W
F. M. Pennebaker, 773-42A
W. R. Wilmarth, 773-A
Records Administration (EDWS)
B. T. Butcher, 773-43A
D. A. Crowley, 773-43A
G. P. Flach, 773-42A
L. L. Hamm, 703-41A
R. A. Hiergesell, 773-43A
K. M. Kostelnik, 730-4B
M. A. Phifer, 773-42A
R. R. Sietz, 773-43A
F. G. Smith, 703-41A
G. A. Taylor, 773-43A