PPR650.1916 Exhibits

A. HGM Wetland Class Scripts used in NASIS

HGM Wetland Class - Prairie Pothole Region

PARAMETER variable1 CHARACTER PROMPT "Area Symbol (e.g. ND001 OR ND*)".
BASE TABLE component.
EXEC SQL
SELECT areasymbol, liid, nationalmusym, musym, muiid, mustatus, imapunitiid, mukey, dmuiid, 
coiid, compname, localphase, comppct_r, hydricrating 
FROM REAL area, REAL legend, imapunit, REAL mapunit, correlation, REAL datamapunit, 
component 
WHERE 
areasymbol IMATCHES variable1 AND 
JOIN area TO legend AND 
JOIN imapunit TO mapunit AND 
JOIN mapunit TO correlation AND 
JOIN correlation TO datamapunit AND 
JOIN datamapunit TO component AND 

#Option to remove the "PARAMETER" for single execution of this report on the 4 target PPR states.
# (area.areasymbol IMATCHES "ND*" OR 
# area.areasymbol IMATCHES "SD*" OR 
# area.areasymbol IMATCHES "MN*" OR 
# area.areasymbol IMATCHES "IA*") AND 
legend.legendsuituse IN ("current wherever mapped") AND 
correlation.repdmu IN ("yes") AND 
imapunit.mustatus IN ("correlated") AND 
component.hydricrating IN ("yes"); 
SORT BY liid, musym SYM, comppct_r DESC, compname LEX.

EXEC SQL
SELECT lsgd.geomfeatid, lsft.geomftname, lsgf.geomfnamep 
FROM component, OUTER (cogeomordesc lsgd), REAL geomorfeattype lsft, REAL geomorfeat 
lsgf 
WHERE 
JOIN component TO lsgd AND 
JOIN lsgd TO lsgf AND 
JOIN lsft TO lsgf AND 
lsft.geomftname IN ("Landscape") AND 
lsgd.rvindicator IN ("yes"); 
SORT BY lsgd.geomfeatid DESC 
AGGREGATE COLUMN lsft.geomftname LAST, lsgf.geomfnamep LAST.

EXEC SQL
SELECT lfgd.geomfeatid, lfft.geomftname, lfgf.geomfnamep, shapeacross, shapedown
FROM component, OUTER (cogeomordesc lfgd), REAL geomorfeattype lfft, REAL geomorfeat lfgf, cosurfmorphss
WHERE
JOIN component TO lfgd AND
JOIN lfgd TO lfgf AND
JOIN lfft TO lfgf AND
JOIN lfgd TO cosurfmorphss AND
  lfft.geomftname IN ("Landform") AND
  lfgd.rvindicator IN ("yes");
SORT BY lfgd.geomfeatid DESC
AGGREGATE COLUMN lfft.geomftname LAST, lfgf.geomfnamep LAST.

DERIVE sodium1 FROM low USING "MLRA10_StPaul" : "SAR MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)".
DERIVE gyp FROM low USING "MLRA10_StPaul" : "GYP MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)".
DERIVE charge FROM low USING "MLRA10_StPaul" : "EC MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)".
DERIVE lime FROM low USING "MLRA10_StPaul" : "CaCO3 MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)".
DERIVE flood1 FROM rv USING "MLRA10_StPaul" : "FLOODING FREQUENCY MAX GROWING SEASON (ND)".
DERIVE histic1 FROM rv USING "MLRA10_StPaul" : "HISTIC EPIPEDON THICKNESS (ND)".

DEFINE cname ISNULL (localphase) THEN compname ELSE cname||", "||localphase.
ASSIGN hydricrating CODENAME(hydricrating).

#Sodium.
DEFINE sodium IF (sodium1 >= 5) THEN "sodic" ELSE "not sodic".

#Histic epipedon.
DEFINE histic IF (histic1 >= 20) THEN "organic" ELSE "mineral".

#Carbonates, gypsum, and other soluble salts.
DEFINE salt IF (((lime >= 1) OR (gyp >= 1) OR (charge >= 4.001))
  OR (localphase IMATCHES "*saline*") OR (compname MATCHES "Salt flats")) THEN "present" ELSE "absent".

#Floods during growing season and on flooded landform.
DEFINE flood IF (((flood1 MATCHES "very frequent") OR (flood1 MATCHES "frequent") OR (flood1 MATCHES "occasional") OR (flood1 MATCHES "rare"))
  AND ((lfgf.geomfnamep MATCHES "flood plains") OR (lfgf.geomfnamep MATCHES "flood-plain steps") OR (lfgf.geomfnamep MATCHES "oxbows") OR
    (lfgf.geomfnamep MATCHES "drainageways") OR
    (lfgf.geomfnamep MATCHES "channels"))) THEN "yes" ELSE "no".

#Define fen HGM wetland class.

(PPR210-650-H, May 2022)
DEFINE wetland_class IF ((sodium MATCHES "not sodic") AND (histic MATCHES "organic") AND (flood MATCHES "no") AND (lfgf.geomfnamep MATCHES "fens")) THEN "fen" ELSE "".

#Define organic flat HGM wetland class.
ASSIGN wetland_class IF (((wetland_class == "") AND (wetland_class != "fen")) AND (sodium MATCHES "not sodic") AND (histic MATCHES "organic") AND (flood MATCHES "no")) THEN "organic flat" ELSE wetland_class.

#Define riverine HGM wetland class.
ASSIGN wetland_class IF (((wetland_class == "") AND (wetland_class != "fen") AND (wetland_class != "organic_flat"))) THEN "riverine" ELSE wetland_class.

#Define mineral flat HGM wetland class.
ASSIGN wetland_class IF (((wetland_class == "") AND (wetland_class != "fen") AND (wetland_class != "organic_flat") AND (wetland_class != "riverine")) AND (sodium MATCHES "not sodic") AND (histic MATCHES "mineral") AND (salt MATCHES "absent") AND (flood MATCHES "no") AND (lfgf.geomfnamep MATCHES "flats")) THEN "mineral flat" ELSE wetland_class.

#Define recharge HGM wetland class.
ASSIGN wetland_class IF (((wetland_class == "") AND (wetland_class != "fen") AND (wetland_class != "organic_flat") AND (wetland_class != "riverine") AND (wetland_class != "mineral_flat")) AND (sodium MATCHES "not sodic") AND (histic MATCHES "mineral") AND (salt MATCHES "absent") AND (flood MATCHES "no") AND (lfgf.geomfnamep MATCHES "depressions")) OR (lfgf.geomfnamep MATCHES "stream terraces") OR (lfgf.geomfnamep MATCHES "sand sheets") OR (lfgf.geomfnamep MATCHES "moraines") OR (lfgf.geomfnamep MATCHES "potholes") OR (lfgf.geomfnamep MATCHES "drainageways") OR (lfgf.geomfnamep MATCHES "flood plains") OR (lfgf.geomfnamep MATCHES "flood-plain steps") OR (lfgf.geomfnamep MATCHES "oxbows") OR (lfgf.geomfnamep MATCHES "abandoned channels")

(PPR210-650-H, May 2022)
#Define discharge HGM wetland class.
ASSIGN wetland_class IF (((((wetland_class == "") AND (wetland_class != "fen") AND (wetland_class != "organic_flat") AND (wetland_class != "riverine") AND (wetland_class != "mineral_flat") AND (wetland_class != "recharge")) AND (sodium MATCHES "not sodic") AND (histic MATCHES "mineral") AND (salt MATCHES "present") AND (flood MATCHES "no")(lfgf.geomfnamep MATCHES "depressions") OR (lfgf.geomfnamep MATCHES "rims") OR (lfgf.geomfnamep MATCHES "terraces") OR (lfgf.geomfnamep MATCHES "playas") OR (lfgf.geomfnamep MATCHES "potholes") OR (lfgf.geomfnamep MATCHES "marshes") OR (lfgf.geomfnamep MATCHES "deltas") OR (lfgf.geomfnamep MATCHES "delta plains") OR (lfgf.geomfnamep MATCHES "outwash plains") OR (lfgf.geomfnamep MATCHES "lakeshores") OR (lfgf.geomfnamep MATCHES "till-floored lake plains") OR (lfgf.geomfnamep MATCHES "collapsed ice-walled lakebeds") OR (lfgf.geomfnamep MATCHES "beaches") OR (lfgf.geomfnamep MATCHES "drainageways") OR (lfgf.geomfnamep MATCHES "flats")(compname MATCHES "Salt flats")) THEN "discharge" ELSE wetland_class.

#Define slope HGM wetland class.
ASSIGN shapeacross CODENAME(shapeacross).
ASSIGN shapetodown CODENAME(shapetodown).
ASSIGN wetland_class IF (((((wetland_class == "") AND (wetland_class != "fen") AND (wetland_class != "organic_flat") AND (wetland_class != "riverine") AND (wetland_class != "mineral_flat") AND (wetland_class != "recharge") AND (sodium MATCHES "not sodic") AND (histic MATCHES "mineral") AND (salt MATCHES "present") AND (flood MATCHES "no")(shapedown MATCHES "concave") OR (shapedown MATCHES "concave") AND (shapeacross MATCHES "linear") OR (shapedown MATCHES "concave") AND (shapeacross MATCHES "convex") OR (shapedown MATCHES "concave") AND (shapeacross MATCHES "concave") OR THEN "discharge" ELSE wetland_class.

(PPR210-650-H, May 2022)
(shapedown MATCHES "linear") AND (shapeacross MATCHES "concave") OR (lfgf.geomfnamep MATCHES "seeps") THEN "slope" ELSE wetland_class.

#Exclude sodium affected soils. Sodium alters the hydrology via dispersion and its affects on water infiltration require an on-site investigation.
ASSIGN wetland_class IF (((wetland_class == "") AND (wetland_class != "fen") AND (wetland_class != "organic_flat") AND (wetland_class != "riverine") AND (wetland_class != "mineral_flat") AND (wetland_class != "recharge") AND (wetland_class != "discharge") AND (wetland_class != "slope")) AND (sodium MATCHES "sodic")) THEN "sodic" ELSE wetland_class.

PAGE WIDTH UNLIMITED LENGTH UNLIMITED.

TEMPLATE alpha SEPARATOR "|" AT LEFT
FIELD WIDTH UNLIMITED SEPARATOR "", FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED, FIELD WIDTH UNLIMITED.

SECTION A
HEADING
USING alpha

(PPR210-650-H, May 2022)
"sodium",
"histic",
"salts",
"floods?",
"wetland_class",
"coiid",
"mukey".
END SECTION.

SECTION B
DATA USING alpha
areasymbol,
nationalmusym,
musym,
compname,
localphase,
comppct_r,
hydricrating,
lsft.geomftname,
lsgf.geomfnamep,
lfft.geomftname,
lfgf.geomfnamep,
sodium,
histic,
salt,
flood,
wetland_class,
coiid NO COMMA ,
mukey NO COMMA .

END SECTION.
SAR MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)

BASE TABLE component.

# Get maximum SAR in any horizon in 0-50 cm range below organic duff and above restriction.
EXEC SQL
SELECT hzdept_r, sar_l, sar_r, sar_h
FROM component, chorizon
WHERE
JOIN component TO chorizon;
SORT BY hzdept_r
AGGREGATE COLUMN hzdept_r NONE, sar_l NONE, sar_r NONE, sar_h NONE.

# Determine the thickness of organic duff and depth to RESTRICTIVE LAYER.
DERIVE o_thickness FROM rv USING "MLRA10_StPaul":"SURFACE ORGANIC DUFF HORIZONS THICKNESS (ND)".
DERIVE depth FROM rv USING "NSSC Pangaea":"DEPTH TO FIRST RESTRICTIVE LAYER".

ASSIGN sar_l IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE sar_l.
ASSIGN sar_h IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE sar_h.
ASSIGN sar_r IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE sar_r.

# Find minimum of restriction depth and 50 cm
DEFINE min_depth depth < 51 AND NOT ISNULL(depth)? depth : 51.
DEFINE in_range ISNULL(hzdept_r)? hzdept_r : ((hzdept_r < min_depth)? 1 : 0).

# Find the sodium adsorption ratio values in the min_depth.
DEFINE low ARRAYMAX(LOOKUP(1, in_range, sar_l)).
DEFINE high ARRAYMAX(LOOKUP(1, in_range, sar_h)).
DEFINE rv ARRAYMAX(LOOKUP(1, in_range, sar_r)).
GYP MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)

BASE TABLE component.

# Get maximum gypsum in any horizon in 0-50 cm range below organic duff and above restriction.
EXEC SQL
SELECT hzdept_r, gypsum_l, gypsum_r, gypsum_h
FROM  component, chorizon
WHERE
JOIN component TO chorizon;
SORT BY hzdept_r
AGGREGATE COLUMN hzdept_r NONE, gypsum_l NONE, gypsum_r NONE, gypsum_h NONE.

# Determine the thickness of organic duff and depth to RESTRICTIVE LAYER.
DERIVE o_thickness FROM rv USING "MLRA10_StPaul":"SURFACE ORGANIC DUFF HORIZONS THICKNESS (ND)".
DERIVE depth FROM rv USING "NSSC Pangaea":"DEPTH TO FIRST RESTRICTIVE LAYER".

ASSIGN gypsum_l IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE gypsum_l.
ASSIGN gypsum_h IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE gypsum_h.
ASSIGN gypsum_r IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE gypsum_r.

# Find minimum of restriction depth and 50cm
DEFINE min_depth depth < 51 AND NOT ISNULL(depth)? depth : 51.
DEFINE in_range ISNULL(hzdept_r)? hzdept_r : ((hzdept_r < min_depth)? 1 : 0).

# Find the gypsum values in the min_depth.
DEFINE low ARRAYMAX(LOOKUP(1, in_range, gypsum_l)).
DEFINE high ARRAYMAX(LOOKUP(1, in_range, gypsum_h)).
DEFINE rv ARRAYMAX(LOOKUP(1, in_range, gypsum_r))
EC MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICTION (ND)

BASE TABLE component.

# Get maximum EC in any horizon in 0-50 cm range below organic duff and above restriction.
EXEC SQL
SELECT hzdept_r, ec_l, ec_r, ec_h
FROM component, chorizon
WHERE
JOIN component TO chorizon;
SORT BY hzdept_r
AGGREGATE COLUMN hzdept_r NONE, ec_l NONE, ec_r NONE, ec_h NONE.

#Determine the thickness of organic duff and depth to RESTRICTIVE LAYER.
DERIVE o_thickness FROM rv USING "MLRA10_StPaul"."SURFACE ORGANIC DUFF HORIZONS THICKNESS (ND)".
DERIVE depth FROM rv USING "NSSC Pangaea"."DEPTH TO FIRST RESTRICTIVE LAYER".

ASSIGN ec_l IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE ec_l.
ASSIGN ec_h IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE ec_h.
ASSIGN ec_r IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE ec_r.

#Find minimum of restriction depth and 50cm
DEFINE min_depth depth < 51 AND NOT ISNULL(depth)? depth : 51.
DEFINE in_range ISNULL(hzdept_r)? hzdept_r : ((hzdept_r < min_depth)? 1 : 0).

#Find the electrical conductivity values in the min_depth.
DEFINE low ARRAYMAX(LOOKUP(1, in_range, ec_l)).
DEFINE high ARRAYMAX(LOOKUP(1, in_range, ec_h)).
DEFINE rv ARRAYMAX(LOOKUP(1, in_range, ec_r)).

(PPR210-650-H, May 2022)
**CaCO3 MAX IN DEPTH 0-50 CM BELOW DUFF, ABOVE RESTRICT (ND)**

BASE TABLE component.

# Get maximum CaCO3 in any horizon in 0-50 cm range below organic duff and above restriction.
EXEC SQL
SELECT hzdept_r, CaCO3_l, CaCO3_r, CaCO3_h
FROM component, chorizon
WHERE
JOIN component TO chorizon;
SORT BY hzdept_r
AGGREGATE COLUMN hzdept_r NONE, CaCO3_l NONE, CaCO3_r NONE, CaCO3_h NONE.

# Determine the thickness of organic duff and depth to RESTRICTIVE LAYER.
DERIVE o_thickness FROM rv USING "MLRA10_StPaul":"SURFACE ORGANIC DUFF HORIZONS THICKNESS (ND)".
DERIVE depth FROM rv USING "NSSC Pangaea":"DEPTH TO FIRST RESTRICTIVE LAYER."

ASSIGN CaCO3_l IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE CaCO3_l.
ASSIGN CaCO3_h IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE CaCO3_h.
ASSIGN CaCO3_r IF hzdept_r > o_thickness + 50 OR hzdept_r < o_thickness THEN NULL ELSE CaCO3_r.

# Find minimum of restriction depth and 50cm
DEFINE min_depth depth < 51 AND NOT ISNULL(depth)? depth : 51.
DEFINE in_range ISNULL(hzdept_r)? hzdept_r : ((hzdept_r < min_depth)? 1 : 0).

# Find the calcium carbonate equivalent values in the min_depth.
DEFINE low ARRAYMAX(LOOKUP(1, in_range, CaCO3_l)).
DEFINE high ARRAYMAX(LOOKUP(1, in_range, CaCO3_h)).
DEFINE rv ARRAYMAX(LOOKUP(1, in_range, CaCO3_r)).
FLOODING FREQUENCY MAX GROWING SEASON (ND)

BASE TABLE component.

# Get the longest flooding frequency class.
EXEC SQL
SELECT flodfreqcl
FROM component, comonth
WHERE 
JOIN component TO comonth AND 
(flodfreqcl NOT IN("none")) AND 
((taxtempregime IN ("mesic") AND 
comonth.month IN("mar", "apr", "may", "jun", "jul", "aug", "sep", "oct"))) OR 
(taxtempregime IN("frigid") AND 
comonth.month IN("apr", "may", "jun", "jul", "aug", "sep"));
AGGREGATE COLUMN flodfreqcl UNIQUE.

DEFINE flooding CODENAME(flodfreqcl).
DEFINE duration flooding MATCHES "very frequent" ? 1 : flooding MATCHES "frequent" ? .8 
: flooding MATCHES "occasional" ? .6 : flooding MATCHES "rare" ? .4 : flooding 
MATCHES "very rare" ? .2 : flooding MATCHES "none" ? .1 : 1/0.
DEFINE longest ARRAYMAX(duration).
DEFINE rv LOOKUP (longest, duration, flooding).
HISTIC EPIPEDON THICKNESS (ND)

BASE TABLE component.

EXEC SQL
SELECT hzdepb_l low, hzdepb_r rv, hzdepb_h high
FROM component
INNER JOIN chorizon ON chorizon.coiidref=component.coiid
INNER JOIN chtexturegrp ON chtexturegrp.chiidref=chorizon.chiid AND
   chtexturegrp.rvindicator = 1
INNER JOIN chtexture ON chtexture.chtgiidref=chtexturegrp.chtgiid
AND ((desgnmaster IN ("O", "O'", "O"')) OR (hzname IN ("O*"))) AND
   ((lieutex IN ("mpm")) OR
    (lieutex IN ("mpt")) OR
    (lieutex IN ("muck")) OR
    (lieutex IN ("peat")) OR
    (lieutex IN ("spm")) OR
    (lieutex IN ("udom")) OR
    (lieutex IN ("pdom")) OR
    (lieutex IN ("hpm")))

AGGREGATE COLUMN low MAX, rv MAX, high MAX.

ASSIGN low IF ISNULL(low) THEN 0 ELSE low.
ASSIGN rv IF ISNULL(rv) THEN 0 ELSE rv.
ASSIGN high IF ISNULL(high) THEN 0 ELSE high.
B. NASIS Query used to Acquire Organic Soil Properties


```
SELECT legend.areasymbol, mapunit.mukey, mapunit.musym, component.cokey, 
component.compname, component.comppct_r, chorizon.chkey, chorizon.hzname, 
chttexturegrp.texture, 
chorizon.hzdept_r, chorizon.hzdepb_r, 
ISNULL(chorizon.ksat_r, -9.9) as 'ksat_r', 
ISNULL(chorizon.om_r, -9.9) as 'om_r', 
ISNULL(chorizon.dbthirdbar_r, -9.9) as 'dbthirdbar_r', 
ISNULL(chorizon.wsatiated_r, -9.9)/100.0 as 'wsatiated_r', 
ISNULL(chorizon.wthirdbar_r, -9.9)/100.0 as 'wthirdbar_r', 
ISNULL(chorizon.wfifteenbar_r, -9.9)/100.0 as 'wfifteenbar_r' 
FROM (legend INNER JOIN (mapunit 
INNER JOIN component ON mapunit.mukey = component.mukey) 
ON legend.lkey = mapunit.lkey) 
INNER JOIN chorizon ON component.cokey = chorizon.cokey 
INNER JOIN chttexturegrp ON chorizon.chkey = chttexturegrp.chkey 
WHERE ((legend.areasymbol like 'IA%') OR (legend.areasymbol like 'MN%') 
OR (legend.areasymbol like 'ND%') OR (legend.areasymbol like 'SD%')) 
AND (component.comppct_r>=10) 
AND (chttexturegrp.rvindicator = 'Yes') AND (mapunit.musym <> 'W') 
AND (chorizon.hzname LIKE 'Oi')) 
ORDER BY mapunit.mukey, component.cokey, chorizon.hzdept_r
```
C. Lateral Effect Calculation Script

"

Created on Tue Dec 15 12:48:34 2020

@author: Jason.Roth
@title: Environmental Engineer WQQT
@affiliation: USDA-NRCS WNTSC
@email:jason.roth@usda.gov

DESC: Script containing suite of functions and calls to produce
a lateral effect distance for the purposes of wetlands conservation

import os
import requests as req
import numpy as np
import math
import sys

def make_file_hdr(names, units=0, delim=" "):  
    ""
    reads in a tab delimited data file

    Parameters
    ---------
    names : list, list of strings of header names
    units : units, list of strings of corresponding units
    delim : str, delimeter character

    Returns
    ------
    h : str, containing file header.
h = str(names[0])
names.pop(0)
for n in names:
    h+="{0} {1}".format(delim, n)
if len(units)>=len(names):
    h+="\n{0}".format(str(units[0]))
    units.pop(0)
    for u in units:
        h+="{0} {1}".format(delim, u)
    return h

def read_data_file(file_path, skp_lin=0):
    """
    reads in a tab delimited data file
    Parameters
    ----------
    file_path : str, path to data file
    skp_lin : int, number of lines to skip at the beginning of the file
    Returns
    ------
    d : list, list containing contents of file
    """
    with open(file_path, 'r') as f:
        i=0
        d = []
        for l in f.readlines():
            if i >= skp_lin:
                d.append(l.split("\t"))
            i+=1
def calc_run_count(comp_count, run_data):
    
    Calculates the number of lateral effect calculations for this batch.

    Parameters
    ----------
    comp_count : int, number of components
    run_data : dict, dictionary of lists of parameters to run calcs for

    Returns
    -------
    rf : int, number of run permutations

    rf = 1
    for k in run_data.keys():
        if k != "states":
            rf *= len(run_data[k])
    rf*=comp_count
    return rf

def read_run_info(run_file):
    
    Reads set of inputs for which to calculate lateral effects for.

    Parameters
    ----------
    run_file : str, path to properly formatted text file with input parameters

    Returns
    -------

(PPR210-650-H, May 2022)
run_par : dict, all runfile parameters

```python
with open(run_file) as f:
    lines = f.readlines()
## dictionary for raw run data
run_par = {}
## dictionary for unique permutations of run data
runs = {}
## temporary storage for run data
x=[]
## integer key for dictionary of run unique run parameters
i = 0
for l in lines:
    l = l.strip()
    if l[0] != "#":
        p, d = l.split(":")
        d = d.strip().split(",")
        if p == "states":
            x = [s.strip() for s in d]
        else:
            x = [float(f.strip()) for f in d]
    # ensure only unique values in each parameter set
run_par[p] = list(set(x))

for ddn_i in run_par['drawdown_init']:
    for ddn_f in run_par['drawdown_final']:
        for d_dia in run_par['drain_diam']:
            for d_dep in run_par['drain_depth']:
                for b_dep in run_par['barrier_depth']:
                    for ss in run_par['surf_storage']:
                        for t in run_par['drawdown_time']:
                            ## enforce physical constraints.
                            ## consider adding ratio of d_dep:b_dep
                            (PPR210-650-H, May 2022)
```
if ((0 <= ddn_i < ddn_f < d_dep < b_dep < 15) 
    and t > 0) 
    and ((ss == 0 and ddn_i > 0) or 
    (ss >= 0 and ddn_i == 0)):
    runs[i]= {'drawdown_init': ddn_i, 
    'drawdown_final': ddn_f, 
    'drain_diam': d_dia, 
    'drain_depth': d_dep, 
    'barrier_depth': b_dep, 
    'surf_storage': ss, 
    'drawdown_time': t} 
    i+=1
else:
    print("no go")
return run_par['states'], runs

def qry_sda(sql):

    """
    requests soil data from sda web service
    For documentation of using web services for SDA see
    https://sdmdataaccess.nrcs.usda.gov/WebServiceHelp.aspx#RunQuery
    """

    url = "https://SDMDataAccess.sc.egov.usda.gov/Tabular/post.rest"

    # sda webservice url
    url = "https://SDMDataAccess.sc.egov.usda.gov/Tabular/post.rest"

    # data formate

    (PPR210-650-H, May 2022)
tab_fmt = "JSON+COLUMNNAME"

## parameter dictionary to pass with post request
params = {"QUERY":sql, "FORMAT":tab_fmt}

## execute the post request
x = req.post(url, data=params)

## extract rows of data row from request
sda_data = x.json() ['Table'][1:]

return sda_data

def qry_rosetta(r_input, r_version):
    """
    request to retrieve rosetta estimates of soil properties
    from Todd Skaggs' (ARS) web-service'
    For documentation of using web services for ROSETTA see
    https://www.handbook60.org/home/
    
    Parameters
    --------
    r_input : list, horizon level texture information for ROSETTA
    r_version : int, rosetta version to use
       1: original schaap version
       2: nrcs version
       3: zhang n schaap 2017
    
    Returns
    ------
    r : list, returned soil hydraulic properties for all horizons
    """

    rosetta_url = f"http://www.handbook60.org/api/v1/rosetta/{r_version}"
    print("Requesting Rosetta estimates ...")
r = req.post(rosetta_url, json={"X": r_input})
if not r.ok:
    print(f"Error!\nStatus code: {r.status_code}\nMessage:\n{r.text}")
sys.exit()

r.json()['_van_genuchten_params']
return r.json()['_van_genuchten_params']

def organic_ptf(dat):
    
    @author- Jason Roth WME NRCS-MN. jason.roth@usda.gov
    This function populates soil hydraulic parameters necessary for
calculation of lateral effects in organic soils.

    Saturated hydraulic conductivity, saturated and residual water contents,
    and bulk densities are populated using a lookup table of these values
    based on soil texture. Values were calculated from analyses of these
    parameters by texture across the PPR.

    PTFs to generate VG hydraulic parameters for organic soils
    adapted from Liu and Lennartz, 2018
    All organic PTFs require bulk density

    Future Dev: consider adding refinement based on bulk density as
    shown in original publication

Parameters
----------
data- numpy array containing soil data

Returns
-------
dat - numpy array containing soil data

(PPR210-650-H, May 2022)
## dictionary to crosswalk database textures to assignments

```python
org_txt_map = {
    "CE": "MUCK",
    "FLV-": "MPT",
    "FL-": "MPT",
    "HB-M": "MUCK",
    "HPM": "MUCK",
    "MARL": "MUCK",
    "MK-P": "MPT",
    "MK-S": "MUCK",
    "MPM": "MPT",
    "MPT": "MPT",
    "MUCK": "MUCK",
    "PEAT": "PEAT",
    "SP": "MUCK",
    "SPM": "PEAT",
    "SR": "MUCK",
    "STX-": "MPT",
    "SR-MUCK FS": "MUCK",
    "COP-": "MUCK"
}
```

## dictionary to reference hydraulic paramters to assign to ea.
## designated texture class
## Ks in log([cm/d]), RhoB in [g/cm^3], water contents in [cm3/cm3]

```python
org_prams = {
    "MUCK": {
        "Ks": 2.33,
        "ThetaS": 0.64,
        "ThetaR": 0.17,
        "RhoB": 0.26
    },
    "MPT": {
        "Ks": 2.90,
        "ThetaS": 0.65,
        "ThetaR": 0.14,
        "RhoB": 0.16
    },
    "PEAT": {
        "Ks": 3.41,
        "ThetaS": 0.67,
        "ThetaR": 0.09,
        "RhoB": 0.11
    }
}
```

## conversion factor for um/s -> cm/d

```python
ums2cmd = 24.*3600/10**4
```

## get the names of horizons that we want to populate in this process

```python
org_key = org_txt_map.keys()
```

## iterate over all horizons

```python
for i in range(dat.shape[0]):
    ## check if the texture is contained in textures to re-assign
t = dat[i]['texture']
    if t in org_key:
        ## get the re-assignment value
        txt = org_txt_map[t]
        ## calculate average horizon depth to be used in VG n param calc
        d = (dat[i]['hzdepb_r'] + dat[i]['hzdept_r'])/2.

        ## store bulk density in var. for readability
        if dat[i]['dbthirdbar_r'] < 0:
```

(PPR210-650-H, May 2022)
dat[i]['dbthirdbar_r'] = org_prams[txt]['RhoB']
bd = dat[i]['dbthirdbar_r']

## assign hydraulic conductivity to this soil
if dat[i]['ksat_r'] < 0:
    ## NOTE Value in lookup dictionary is already log x-formed
    dat[i]['Ks'] = org_prams[txt]['Ks']
else:
    ## convert NASIS value from um/s to cm/d, log xform and store
    dat[i]['Ks'] = math.log10(dat[i]['ksat_r']*ums2cmd)

## assign saturated water content to this soil
if dat[i]['wsatiated_r'] < 0:
    dat[i]['ThetaS'] = org_prams[txt]['ThetaS']
else:
    dat[i]['ThetaS'] = dat[i]['wsatiated_r']

## assign residual water content to this soil
if dat[i]['wthirdbar_r'] < 0:
    dat[i]['ThetaR'] = org_prams[txt]['ThetaR']
else:
    dat[i]['ThetaR'] = dat[i]['wthirdbar_r']

## Van Genuchten parameters need to be generated for all organics
## calculate and store VG log('alpha') parameter for this soil
## from Liu and Lennartz, 2018 Table 4
dat[i]['alpha'] = (0.326-9.315*bd+10.420*bd**2-0.014*d)
## calculate and store VG log('n') to this soil
## from Liu and Lennartz, 2018 Table 4
dat[i]['n'] = (0.153-0.422*bd+0.45*bd**2)

return dat

def calc_soil_conductivity_nrcs(sol_arr, drn_dep, bar_dep, top_dep=0.0):
    
    (PPR210-650-H, May 2022)
calculated hydraulic conductivity above and below drain

assumes soil horizon depths are in cm, hydraulic conductivity log(cm/d)

Parameters

-------------
soll_arr: np recarray, containing depth (cm) and Ks log(cm/d) data for
  a soil component. Array must have named
    fields ['hzdepb_r', 'Ks']
drn_dep: float, drain depth (cm)
bar_dep: float, barrier depth (cm)

Returns

-------------
ska: float, hydraulic conductivity (cm/d) above drain depth
skb: float, hydraulic conductivity (cm/d) below drain depth

if drn_dep >= bar_dep:
    bar_dep = drn_dep + 1.0

old_bot_dep = top_dep

# state variable for calculating above (1) or below (2) drain
lyr = 1
ksat = 0
lyr_lst = range(sol_arr.shape[0])

sol_arr.sort(order='hzdepb_r')

# figure out what horizon we start in
i = 0
while top_dep > sol_arr['hzdepb_r'][i]:

(PPR210-650-H, May 2022)
i+=1

for i in range(i, sol_arr.shape[0]):
    bot_dep = sol_arr['hzdepb_r'][i]
    k = sol_arr['Ks'][i]

    # determine if new bottom depth is greater than drain depth and
    # need to increment to conductivity below drain and or this is the
    # last depth entry for this component and need to extrapolate values
    # downward to confining layer.
    
    ## this catches condition where horizon intersects drain depth or
    ## process runs out of soils data and needs to be extrapolated down
    ## to the barrier depth.
    if (bot_dep >= drn_dep and lyr == 1) or i == lyr_lst[-1]:
        ## Both of these ifs can trigger sequentially if drain depth
        ## is below the bottom depth of the deepest soil horizon
        ## case - horizon intersects drain depth
        if lyr == 1:
            ## incremental layer thickness
            thk = (drn_dep - old_bot_dep)
            ## calculate total layer thickness down to drain depth
            lyr_thk = drn_dep
            ## undo log-xform and multiply by incremental thickness and
            ## add to the running sum of products
            ## std_dev = k_u = 0
            ksat += thk * 10**(k)
            ## divide total sum product by total layer thickness to get
            ## depth weighted average of horizontal conductivity
            ka = ksat/lyr_thk
            ## increment the layer counter and calculate for first
            ## increment below drain and
            lyr = 2
            ## incremental layer thickness
            thk = (bot_dep - drn_dep)
## undo log-xform and multiply by incremental thickness and
## add to the running sum of products
## std_dev = k_u = 0
ksat = thk * 10**(k)
old_bot_dep = bot_dep
## case - ran out of soils data
if i == lyr_lst[-1]:
    ## check if last soils increment is less than the barrier depth
    ## in this case we need to extrapolate downward to the
    ## barrier depth. There has been discussion of using a decay
    ## factor based on the assumption that conductivity is reduced
    ## at depth due to consolidation
    if bot_dep < bar_dep:
        bot_dep = bar_dep
        ksat += (bot_dep - old_bot_dep)*10**(k)
        lyr_thk = (bot_dep - drn_dep)
        kb = ksat/lyr_thk
        old_bot_dep = 0
else:
    thk = (bot_dep - old_bot_dep)
    ## undo log-xform and multiply by incremental thickness
    ksat += thk * 10**(k)
old_bot_dep = bot_dep

return ka, kb

def calc_soil_porosity_nrcs(sol_arr, wtb_ini=0.0, wtb_fin=1.0, inc=1.0):
    """
    NRCS-EFH Ch 19-60
    calculate drainable porosity as function of head between 2 drawdown depths
    i.e. water retention drawdown curve using Van Genuchten relationships.
    """

    Parameters

    (PPR210-650-H, May 2022)
----------
soll_arr: np recarray, containing depth (cm) and Ks log(cm/d) data for
a soil component. Array must have named
fields ['hzdepb_r', 'ThetaS', 'ThetaR', 'alpha', 'n']

wtb_ini: float, initial water table depth (cm)

wtb_fin: float, final water table depth (cm)

Returns
------
f: float, drainable porosity (in/in) between drawdown depths

wtb = wtb_ini

hed = 0

## initialize drainable porosity at 0
f = 0
## set soil horizon index variable to 0
a = 0
## ensure soil values are sorted.
sol_arr.sort(order='hzdepb_r')

## make sure calculation begins in horizon containing the initial
## water table depth
while sol_arr['hzdepb_r'][a] < wtb_ini and a < sol_arr.shape[0]-1:
    a += 1
## set initial temp value of bot to get into loop...sloppy.
bot = 1.0

## iterate over depth btwn initial and final water table in 1cm increments
while wtb <= wtb_fin:
if (wtb > bot and a < sol_arr.shape[0]-1) or wtb==wtb_ini:
    ## if this depth is greater than the horizon depth increment "a"
    ## so functions reference correct soil parameters for this depth
    if a < sol_arr.shape[0]-1 and wtb >= bot:
        a += 1
    ## populate parameters for VG equation for this horizon
    bot = min(sol_arr['hzdepb_r'][a], wtb_fin)
    ths = sol_arr['ThetaS'][a]
    thr = sol_arr['ThetaR'][a]
    enn = sol_arr['n'][a]
    alfa = sol_arr['alpha'][a]
    ## transform from log values
    enn = 10**enn
    alfa = 10**alfa
    emm = 1./enn
    ## calculate total water drained to this depth
    ## theta_s minus VG water retention at this depth : ch 19, eqn 19-4
    if wtb + inc < wtb_fin:
        hed = wtb_fin - (wtb+inc/2.)
    else:
        hed = (wtb_fin - wtb)/2.

    if (ths - (thr + (ths - thr)) <= 0 or \
        (1.0 + abs(alfa*hed)**enn)**emm) <= 0:
        print("halt")

    f += ths - (thr + (ths - thr) / 
                (1.0 + abs(alfa*hed)**enn)**emm)

    wtb+=inc
    ## normalize drainable porosity to depth drained to get average
    ## cm/cm drained for the drawdown depth
    f/=(wtb_fin-wtb_ini)
# if drainable porosity is greater than 0.5 this method is not appropriate
if f > 0.5:
    f = -9.9
if np.isnan(f):
    f = -9.9

return f

def calc_vansch(wtb_ini, wtb_fin, drn_dep, drn_dia, bar_dep, hcn_abv, hcn_bel, drn_por, sur_str, tim_per):
    """
    DESC : calculate a lateral effect using the VanS method for a specified
time period. adapted from:
Parameters
--------
wtb_ini: float, initial water table depth below grade (ft)
wtb_fin: float, final water table depth below grade (ft)
drn_dep: float, drain depth below grade (ft)
drn_dia: float, drain diam (in)
bar_dep: float, depth below grade to impervious barrier (ft)
ecn_abv: float, hydraulic conductivity above drain (ft/d)
ecn_bel: float, hydraulic conductivity below drain (ft/d)
drn_por: float, drainable porosity (-/)
sur_str: float, surface storage of water (ft)
tim_per: float, hydraulic conductivity above drain (ft/d)
    """

(PPR210-650-H, May 2022)
Returns
------
le: float, lateral effect, half the drain spacing that results in specified
drawdown over the specified time period (ft).

## assign some constants
pie = 3.1416
tol = 0.01
eff_dep = bar_dep - drn_dep
## get effective radius from function
eff_rad = get_eff_rad(drn_dia)
## set some interim variables
aye = bar_dep - drn_dep
drn_dep = drn_dep - wtb_ini
drn_fin = drn_dep - wtb_fin
eff_por = drn_por + (sur_str /(12.*(emm_ini - emm_fin)))
## calculate representative hcon
eff_hcn = ((hcn_abv * emm_ini) + (hcn_bel * eff_dep)) / (emm_ini + eff_dep)
## get initial ess_pri using the actual depth of drain
num = 9. * eff_hcn * tim_per * eff_dep
den = eff_por * (math.log(emm_ini * (2. * aye + emm_fin)) -
    math.log(emm_fin * (2. * aye + emm_ini)))

ess = 1000

if num > 0 and den > 0:
    ess_pri = math.sqrt(num / den)
    ## initialize 'err'>tol' so we commit to while loop atleast 1x
    err = tol * 1.1
    ## initialize 'err'>tol' so we commit to while loop atleast 1x
    ess = 0
    ## loop while error is greater than
    ## specified tolerance (NRCS recommends 10% tolerance)
    while (err > tol):
        ## determine eff_dep
        if aye / ess_pri <= 0.3:
eff_dep = aye / (1. + (aye / ess_pri) *
          ((8. / pie) * math.log(aye / eff_rad) - 3.4))
else:
    eff_dep = ess_pri * pie /
          (8. * (math.log(ess_pri / eff_rad) - 1.15))
## calculate new spacing for this iteration
num = 9. * eff_hcn * tim_per * eff_dep
     + 8. * (math.log(emm_ini * (2. * eff_dep + emm_fin)) -
               math.log(emm_fin * (2. * eff_dep + emm_ini)))
    / (8. * (math.log(ess_pri / eff_rad) - 1.15))
if num > 0 and den > 0:
    ess_pri = math.sqrt(num / den)
else:
    print(drn_por, hcn_abv, hcn_bel)
    err = abs(ess - ess_pri) / ess_pri
    ess = ess_pri
return ess / 2.0

def get_eff_rad(in_dia):
    """
    DESC: simple lookup of effective radius for a drain based on diameter
    Parameters
    ----------
    in_dia : int, diameter of drain
    Returns
    -------
    Re : float, effective radius of drain
    """
    if in_dia <= 3:
        Re = 0.0115
    elif in_dia > 3 and in_dia <= 4:
        Re = 0.0167
    elif in_dia > 4 and in_dia <= 5:
Re = 0.034
elif in_dia > 5 and in_dia <= 6:
    Re = 0.048
elif in_dia > 6 and in_dia <= 8:
    Re = 0.08
elif in_dia > 8 and in_dia <= 10:
    Re = 0.111
elif in_dia > 10 and in_dia <= 12:
    Re = 0.142
elif in_dia > 12 and in_dia <= 16:
    Re = 0.25
else:
    Re = 1.0
return Re

def check_soils(ptf_data, min_pct=15, maxmin_dep=10., minmax_dep=120.,
                nul_val=-9.9):
    """
    run through soils and make sure that there's enough data and if
    not explain how
    Parameters
    ----------
    ptf_data : numpy recarray, processed soils data needed to calculated
               lateral effects. Expects

    min_pct : int, min component percent for mapunit to be valid

    maxmin_dep : int, maximum minimum depth at which a horizon data are needed
                 to produce a value for this component

    minmax_dep : int, minimum maximum depth at which a horizon data are needed
                 to produce a value for this component

    nul_val : float, the value for missing data
    """

    (PPR210-650-H, May 2022)
Returns

--------

mky_chk : numpy recarray, mukeys indicating whether enough data is present for lateral effect determination

cky_chk : numpy recarray, cokeys indicating whether enough data is present for lateral effect determination

```
mky_type = ['U6', 'U8', 'U8', 'i4']
mky_cols = ['asym', 'mukey', 'musym', 'code']
mky_dt = list(zip(mky_cols, mky_type))

cky_type = ['U6', 'U8', 'U8', 'U8', 'U20', 'i4']
cky_cols = ['asym', 'mukey', 'musym', 'cokey', 'compname', 'code']
cky_dt = list(zip(cky_cols, cky_type))
```

## container for mapunit designations based on analysis

mky_lst = []

cky_lst = []

## get unique mukeys to iterate over

mukeys = np.unique(ptf_data['mukey'])

## get rid of all null data

ptf_data = ptf_data[np.where(
    np.logical_and(ptf_data['Ks'] != nul_val,
    np.logical_and(ptf_data['ThetaR'] != nul_val,
    np.logical_and(ptf_data['ThetaS'] != nul_val,

(PPR210-650-H, May 2022)
## n
##ow compare components for each mukey

```python
for mkey in mukeys:
    pct = 0
    # get unique components for this map unit
    mkey_data = ptf_data[np.where(ptf_data['mukey'] == mkey)]
    if mkey_data.shape[0] > 0:
        asym = mkey_data['areasymbol'][0]
        msym = mkey_data['musym'][0]
        mkey = mkey_data['mukey'][0]
        ## is there any data for the mapunit
        ## iterate over unique components in mapunts and store max result
        cokeys = np.unique(mkey_data['cokey'])
        for ckey in cokeys:
            ## store component soil data
            comp_data = mkey_data[mkey_data['cokey'] == ckey]
            ## is there any data for this component
            if comp_data.shape[0] >= 0:
                cnam = comp_data['compname'][0]
                ckey = comp_data['cokey'][0]
                cpct = comp_data['comppct_r'][0]
                ## sort existing component data by depth
                ## determine if there's top and bottom soil data
                #print("max depth {0}, {1}".format(
                #    comp_data['hzdepb_r'].max(),
                #    cpct))
                if comp_data['hzdept_r'].min() <= maxmin_dep and \
                    comp_data['hzdepb_r'].max() > minmax_dep:
                    pct += cpct
            else:
```

---

(PPR210-650-H, May 2022)
def calc_lateral_effects(ptf_data, run_data):
    
    DESC: calculates lateral effect for unique soil mapunits contained in ptf_data using parameters from run_data and stores the results in led_data.

    Parameters
    ----------
    ptf_data : numpy recarray
        processed soils data needed to calculated lateral effects. Expects column headers of
    led_data : numpy recarray
        calculated lateral effect distance for each mapunit/run parameter permutation
    run_data : Dict
        Dictionary containing all permutations of unique input parameters.

    Returns
led_data : list
    calculated lateral effect distance for each mapunit/run parameter permutation
    [['areasymbol', 'mukey', 'musym', 'cokey', 'compname', 'comppct',
      'drain_depth', 'drain_diam', "surf_storage", "barrier_depth",
      "drawdown_init", "drawdown_final", "drawdown_time",
      "Kh_abv", "Kh_bel", "eff_por", "leff_dist"]]

no_led_data : list
    counties, mapunits and components that didn't have sufficient data for lateral effect calculations
    [['areasymbol', 'mukey', 'musym', 'cokey', 'compname', 'comppct']]

---

i = 0

ft2cm = 12.0*2.54
min_pct = 50.0
## get number of unique components
rukeys = run_data.keys()
pct_inc = 5
rc = 0
rpct = 5
prog_msg = "{0}% done with calculations for {1}"
st = ptf_data['areasymbol'][0][0:2]

null_mkey = []
led_data = []

### check soils for complete data
mky_chk, cky_chk = check_soils(ptf_data, min_pct=50,
    maxmin_dep=10.,
    minmax_dep=120.,
    nul_val=-9.9)
mukeys = np.unique(ptf_data['mukey'])
rcnt = mukeys.shape[0]*len(rukeys)
print("calculating lateral effects for {0}".format(st))
print("total run count is {0}".format(rcnt))

## determine which components for this mapunit
## have sufficient data for LE cal
for r in rukeys:
    rd = run_data[r]
    ddn_i = rd['drawdown_init']
    ddn_f = rd['drawdown_final']
    d_dia = rd['drain_diam']
    d_dep = rd['drain_depth']
    b_dep = rd['barrier_depth']
    ss = rd['surf_storage']
    t = rd['drawdown_time']

for mk in mukeys:
    code = mky_chk[np.where(mky_chk['mukey']==mk)]['code'][0]
    rc += 1
    if rc/rcnt*100.0 > rpct:
        print(prog_msg.format(rpct, st))
        print(rc, rcnt)
        rpct+=pct_inc
    mu_data = ptf_data[np.where(ptf_data['mukey']==mk)]
    asym = mu_data['areasymbol'][0]
    msym = mu_data['"musym"'][0]
    mkey = mu_data['"mukey"'][0]
    if code > 0:
        ## set initial lateral effect for this mapunit to zero
        max_now = 0
        ## iterate over unique map units in data
        ## make sure this is sorted

(PPR210-650-H, May 2022)
tot_pct = 0
cokeys = np.unique(mu_data['cokey'])
for ck in cokeys:
    comp_data = ptf_data[np.where(ptf_data['cokey']==ck)]
    comp_data.sort(order="hzdept_r")
    cnam = comp_data['compname'][0]
    ckey = comp_data['cokey'][0]
    cpct = comp_data['comppct_r'][0]
    ## convert values to cm in f(x) call
    f = calc_soil_porosity_nrcs(comp_data, ddn_i*ft2cm,
                                ddn_f*ft2cm, inc=1.0)
    ## functions expects values in cm
    hka, hkb = calc_soil_conductivity_nrcs(comp_data,
                                            d_depm*ft2cm,
                                            b_depm*ft2cm)
    ## convert values to ft/d
    hka/=ft2cm
    hkb/=ft2cm
    ## final check on data and need for run
    if f > 0.0 and hka > 0.0 and hkb > 0.0:
        ## function expects values in ft and days.
        ## assume constant barrier depth and
        ## initial water table for now
        le = calc_vansch(ddn_i, ddn_f, d_depm, d_dia,
                         b_depm, hka, hkb, f, ss, t)
        le = (int(le/10.0)+1)*10.
        tot_pct += cpct
        ## check if this lateral effect is larger than
        ## the previous max for this mapunit
        if le > max_now:
            max_now = le

if tot_pct >= min_pct:
dat = [asym, mkey, msym, ckey, cnam, cpct,
    d_dep, d_dia, ss, b_dep, ddn_i, ddn_f,
    t, hka, hkb, f, max_now]

else:
    dat = [asym, mkey, msym, 99999999, "unknown", 0,
        d_dep, d_dia, ss, b_dep, ddn_i, ddn_f,
        t, -9.9, -9.9, -9.9, -2]

else:
    dat = [asym, mkey, msym, 99999999, "unknown", 0,
        d_dep, d_dia, ss, b_dep, ddn_i, ddn_f,
        t, -9.9, -9.9, -9.9, -1]

led_data.append(dat)
i+=1

print("fished calculating lateral effects for {0}".format(st))
return led_data, null_mkey

### MAIN FUNCTION

def main():
    ""
    DESC: main control loop

    Parameters
    ---------
    none

    Returns
    -------
    none
    ""
run_file_name = "input.txt"
output_dir = "leff_data"

## formattable strings for state specific data files
sda_file = "{0}_sda_qry.txt"
ptf_file = "{0}_rout.txt"
led_file = "{0}_led.txt"
no_led_file = "{0}_no_led.txt"

## query text
sql_compsWithData = ""
SELECT legend.areasymbol, mapunit.mukey, mapunit.musym,
          component.cokey, component.compname, component.comppct_r,
          chorizon.chkey, chtexturegrp.texture,
          ISNULL(chorizon.ksat_r, -9.9) as 'ksat_r',
          ISNULL(chorizon.wsatiated_r, -9.9)/100.0 as 'wsatiated_r',
          chorizon.hzdept_r, chorizon.hzdepb_r,
          ISNULL(chorizon.sandtotal_r, -9.9) as 'sandtotal_r',
          ISNULL(chorizon.silttotal_r, -9.9) as 'silttotal_r',
          ISNULL(chorizon.claytotal_r, -9.9) as 'claytotal_r',
          ISNULL(chorizon.dbthirdbar_r, -9.9) as 'dbthirdbar_r',
          ISNULL(chorizon.wthirdbar_r, -9.9)/100.0 as 'wthirdbar_r',
          ISNULL(chorizon.wfifteenbar_r, -9.9)/100.0 as 'wfifteenbar_r'
FROM (legend LEFT JOIN
          (mapunit LEFT JOIN
           (component LEFT JOIN
            (chorizon LEFT JOIN chtexturegrp ON chorizon.chkey = chtexturegrp.chkey)
            ON component.cokey = chorizon.cokey)
           ON mapunit.mukey = component.mukey)
      ON legend.lkey = mapunit.lkey)
WHERE legend.areasymbol like '{0}%' AND component.comppct_r>=10
AND chtexturegrp.rvindicator = 'Yes' AND mapunit.musym <> 'W'
AND (chorizon.hzname NOT LIKE 'Cr%')

(PPR210-650-H, May 2022)
ORDER BY mapunit.mukey, component.cokey, chorizon.hzdepb_r
```
sql_allComps = '''
SELECT legend.areasymbol, mapunit.mukey, mapunit.musym,
component.cokey, component.compname, component.comppct_r,
chorizon.chkey
FROM (legend LEFT JOIN
(mapunit LEFT JOIN
(component LEFT JOIN chorizon ON component.cokey = chorizon.cokey)
ON mapunit.mukey = component.mukey)
ON legend.lkey = mapunit.lkey)
WHERE legend.areasymbol like '{0}%' AND component.comppct_r>=10
AND mapunit.musym <> 'W'
ORDER BY mapunit.mukey, component.cokey
'''

## header and datatype for sda query results
sda_data_cols = ['areasymbol', 'mukey', 'musym', 'cokey', 'compname',
 'comppct_r', 'chkey', 'texture', 'ksat_r', 'wsatiated_r',
 'hzdept_r', 'hzdepb_r', 'sandtotal_r', 'silttotal_r',
 'claytotal_r', 'dbthirdbar_r', 'wthirdbar_r', 'wififteenbar_r']
sda_data_type = ['U8', 'U8', 'U10', 'U8', 'U32',
'i2', 'U8', 'U8', 'f4', 'f4',
'i2', 'i2', 'f4', 'f4',
'f4', 'f4', 'f4', 'f4']
sda_data_unit = ['sta_fip', 'key', 'name', 'key', 'name',
'\%', 'key', 'txt', 'um/s', 'cm3/cm3',
'cm', 'cm', '\%', '\%',
'\%', 'g/cm3', 'cm3/cm3', 'cm3/cm3']
sda_hdr = make_file_hdr(sda_data_cols.copy(), sda_data_unit.copy(), delim)
sda_comp_cols = sda_data_cols[0:6]
sda_comp_type = sda_data_cols[0:6]

## this is the order that data will come back
ptf_cols = ['areasymbol', 'mukey', 'musym', 'cokey', 'compname',
    'comppct_r', 'chkey', 'texture', 'hzdept_r', 'hzdepb_r',
    "ThetaR", "ThetaS", "alpha", "n", "Ks"]

ptf_type = ["U8", "U8", "U10", "U8", "U32",
    "i2", "U8", "U8", "i2", "i2",
    "f4", "f4", "f4", "f4", "f4"]

ptf_unit = ["sta_fip", "key", "name", "key", "name",
    ", key", "txt", "cm", "cm",
    "cm3/cm3", "cm3/cm3", "log(cm/d)", "log(1/cm)", "log(cm/d)"]

ptf_hdr = make_file_hdr(ptf_cols.copy(), ptf_unit.copy(), delim)

led_cols = ['areasymbol', 'mukey', 'musym', 'cokey', 'compname', 'comppct_r',
    'drain_depth', 'drain_diam', "surf_storage", "barrier_depth",
    "drawdown_init", "drawdown_final", "drawdown_time",
    "Kh_abv", "Kh_bel", "eff_por", "leff_dist"]

led_type = ["U8", "U8", "U8", "U8", "U32", "i2",
    "f4", "f4", "f4", "f4",
    "f4", "f4", "f4", "f4",
    "f4", "f4", "f4","i4"]

led_unit = ["sta_fip", "key", "name", "key", "name", ", key", "txt", "ft", "in", "in", "ft",
    "ft", "ft", "d",
    "ft/d", "ft/d", "ft/ft", "ft"]

led_hdr = make_file_hdr(led_cols.copy(), led_unit.copy(), delim)
## make a subdirectory to store data in

data_dir = os.path.join(os.getcwd(), output_dir)
if not os.path.exists(data_dir):
    os.mkdir(data_dir)

run_file_path = os.path.join(os.getcwd(), run_file_name)

if os.path.exists(run_file_path):
    states, run_data = read_run_info(run_file_path)

## iterate over each state in the list of states specified in run_file
## tried querying more than one state at a time but POST request fails
for s in states:
    run_type = 0
    ## determine if any of the data exist for this state
    if os.path.exists(os.path.join(data_dir, led_file.format(s))):
        run_type = 3
    elif os.path.exists(os.path.join(data_dir, ptf_file.format(s))):
        run_type = 2
    elif os.path.exists(os.path.join(data_dir, sda_file.format(s))):
        run_type = 1
    else:
        run_type = 0

## if run type < 3 some work needs to be done.
if run_type < 3:
    ## if there is no soil query in data directory need to query
    ## soil data for this state from SDA
    if run_type <= 1:
        ## if soil data already exists import it to run through rosetta
        sda_file_pth = os.path.join(data_dir,sda_file.format(s))
        sda_dt = list(zip(sda_data_cols,sda_data_type))
        if run_type == 1:
            print("Local SDA data found for {0}".format(s))
            print("Importing soil data from " + sda_file.format(s))

(PPR210-650-H, May 2022)
sda_data = read_data_file(sda_file_pth, skp_lin=2)
for i in range(len(sda_data)):
    for j in range(8, 18):
        sda_data[i][j] = float(sda_data[i][j])

else:
    # query SDA using web service
    print("Soil data not found for {0}".format(s))
    print("Querying soil data from SDA")
    sda_data = qry_sda(sql_compsWithData.format(s))

    ## construct a record array from the query results.
    ## rec arrays are much quicker in execution than dataframes
    ## need to convert to tuple so sequential fields of same
    ## data type aren't combined
    arr = np.array([tuple(sd) for sd in sda_data],
                    dtype=sda_dt)

    ## replace all values in ksat_r, wsatiated_r, sandtotal_r
    ## silttotal_r, claytotal_r, dbthirdbar_r, wthirdbar_r
    ## w fifteenbar_r
    for f in ["ksat_r", "wsatiated_r", "sandtotal_r",
              "silttotal_r", "claytotal_r", "dbthirdbar_r",
              "wthirdbar_r", "w fifteenbar_r"]:
        arr[f][np.where(arr[f]<0.0)] = -9.9000

    ## add placeholder entries for soil map units that
    ## didn't return any data from SDA
    ## get all the comps in the state and compare them to
    ## to those with data and fill in the comps that don't
## have data with null

```python
print("Querying component data from SDA")
sda_comp_data = qry_sda(sql_allComps.format(s))
```

```python
no_hzn = []
```

```python
sda_comp_data = qry_sda(sql_allComps.format(s))
```

## make

```python
for scd in sda_comp_data:
    if scd[1] == None:
        x = scd[0:6] + ['99999999'] + ['UKN'] + 2*[-.9] + \
            [0, 10] + 6*[-.9]
        no_hzn.append(x)
        sda_data.append(x)
```

```python
cmp_arr = np.array([tuple(cd) for cd in no_hzn],
                   dtype=sda_dt)
arr = np.append(arr, cmp_arr)
```

## save a copy of the sda query to a file for future ref

```python
np.savetxt(sda_file_pth, arr , fmt="%s", delimiter=delim,
           header=sda_hdr, comments="")
```

```python
del arr
```

if run_type <= 2:
    ## soils data for this state has already been processed
    ptf_file_pth = os.path.join(data_dir, ptf_file.format(s))
    if run_type == 2:
        print("Local PTF data found for {0}".format(s))
        print("Importing PTF data from " +ptf_file.format(s))
        ptf_data = read_data_file(ptf_file_pth, skp_lin=2)
        dt = list(zip(ptf_cols, ptf_type))
        ptf_data = [tuple(i) for i in ptf_data]
ptf_data = np.array(ptf_data, dtype=dt)
else:
    print("PTFs data not found for {0}".format(s))
    print("Processing soils through PTFs")
    ptf_data = qry_rosetta([sd[12:] for sd in sda_data], 2)

    ## need to convert to tuple so sequential fields of same
    ## data type aren't combined

    ## merge sda_data and rosetta output
    temp = []
    for i,j in zip(sda_data, ptf_data):
        temp.append(tuple(i+j))
    dt = list(zip(sda_data_cols+ptf_cols[-5:],
                  sda_data_type+ptf_type[-5:]))
    temp = np.array(temp, dtype=dt)
    temp= organic_ptf(temp)

    ptf_data = temp
    del temp

    ptf_data['Ks'][np.isnan(ptf_data['Ks'])] = -9.9
    ptf_data['ThetaR'][np.isnan(ptf_data['ThetaR'])] = -9.9
    ptf_data['ThetaS'][np.isnan(ptf_data['ThetaS'])] = -9.9
    ptf_data['alpha'][np.isnan(ptf_data['alpha'])] = -9.9
    ptf_data['n'][np.isnan(ptf_data['n'])] = -9.9

    np.savetxt(ptf_file_pth,
                ptf_data[[c for c in ptf_cols]],
                fmt="%s",
                delimiter=delim,
                header=ptf_hdr,
                comments="")
if run_type <= 3:
    led_file_pth = os.path.join(data_dir, led_file.format(s))
    no_led_file_pth = os.path.join(data_dir, no_led_file.format(s))
    led_dt = list(zip(led_cols, led_type))

    #no_led_file_pth = os.path.join(data_dir, no_led_file.format(s))
    if run_type == 3:
        print("Lateral Effect Distances for {0} exist".format(s))
    else:
        print("Lateral Effect not found for {0}".format(s))
        print("Processing Lateral Effects")

        ## calculate lateral effects
    led_data, mky_chk = calc_lateral_effects(ptf_data, run_data)

    arr = np.array([tuple(le) for le in led_data], dtype=led_dt)

    arr.sort(order=['surf_storage', 'mukey', 'drain_depth',
                    'drain_diam'])
    np.savetxt(led_file_pth, arr, fmt="%s",
               delimiter=delim, header=led_hdr, comments="")

    np.savetxt(no_led_file_pth,
               mky_chk[np.where(mky_chk['code']<=0)], fmt="%s",
               delimiter=delim, comments="")

if __name__ == "__main__":
    main()