Appendix H
GIS Tool Python Script

Name: Gridded PMP Tool Python Script

Script Version: 2

Python Version: 2.7

ArcGIS Version: ArcGIS Desktop 10.7.1

Author: Applied Weather Associates

Usage: The tool is designed to be executed within an the ArcMap environment with an open MXD session.

Description:
   This tool calculates PMP depths for a given drainage basin for the specified durations. PMP point values are calculated (in inches) for each grid point (spaced at 90 arc-second intervals) over the project domain. The points are converted to gridded PMP datasets for each duration.

---'''

```
################################################################################
## import Python modules

import sys
import arcpy
import os
import traceback
from arcpy import env
import arcpy.analysis as an
import arcpy.management as dm
import arcpy.conversion as con
import numpy as np
import pandas as pd
from pandas import ExcelFile
import matplotlib.pyplot as plt
from heapq import nlargest


env.overwriteOutput = True                                  # Set overwrite option
env.addOutputsToMap = False

################################################################################
## get input parameters
```

```
basin = arcpy.GetParameter(0)                                    # get AOI Basin Shapefile
home = arcpy.GetParameterAsText(1)                               # get location of 'PMP'
Project Folder
outLocation = arcpy.GetParameterAsText(2)
if arcpy.GetParameter(12) == False:
   locDurations = arcpy.GetParameter(3)                          # get local storm durations
(string)
   genDurations = arcpy.GetParameter(4)                          # get general storm
durations (string)
   tropDurations = arcpy.GetParameter(5)                         # get tropical storm
durations (string)
else:
   locDurations = ('01','02','03','04','05','06','12','24')
   genDurations = ('01','02','03','04','05','06','12','24','48','72')
   tropDurations = ('01','02','03','04','05','06','12','24','48','72')

weightedAve = arcpy.GetParameter(8)                                  # get option to
apply weighted average (boolean)
#outputTable = arcpy.GetParameter(9)                                 # get file path
for basin average summary table
includeSubbasin = arcpy.GetParameter(9)                          # get option add
subbasin averages (boolean)
subbasinIDfield = arcpy.GetParameterAsText(10)                       # Subbasin ID
field from AOI Basin Shapefile
ddChart = arcpy.GetParameter(11)                                 # get option to create
depth-duration chart(boolean)
runTemporal = arcpy.GetParameter(12)                                 # get option to
run temporal distributions (boolean)

dadGDB = home + "\\Input\\DAD_Tables.gdb"                        # location of DAD tables
adjFactGDB = home + "\\Input\\Storm_Adj_Factors.gdb"                  # location of feature
datasets containing total adjustment factors
arcpy.AddMessage("\nDAD Tables geodatabase path:  " + dadGDB)
arcpy.AddMessage("Storm Adjustment Factor geodatabase path:  " + adjFactGDB)

#mxd = arcpy.mapping.MapDocument("CURRENT")
#df = arcpy.mapping.ListDataFrames(mxd)[0]
basAveTables = []                                            # global list of Basin Average
Summary tables

def pmpAnalysis(aoiBasin, stormType, durList):

   ##########################################################################
   ## Create PMP Point Feature Class from points within AOI basin and add fields
   def createPMPfc():
```

```
        arcpy.AddMessage("\nCreating feature class: 'PMP_Points' in Scratch.gdb...")
        dm.MakeFeatureLayer(home + "\\Input\\Non_Storm_Data.gdb\\Vector_Grid", "vgLayer")
# make a feature layer of vector grid cells
        dm.SelectLayerByLocation("vgLayer", "INTERSECT", aoiBasin)                    #
select the vector grid cells that intersect the aoiBasin polygon
        dm.MakeFeatureLayer(home + "\\Input\\Non_Storm_Data.gdb\\Grid_Points", "gpLayer")
# make a feature layer of grid points
        dm.SelectLayerByLocation("gpLayer", "HAVE_THEIR_CENTER_IN", "vgLayer")
# select the grid points within the vector grid selection
        con.FeatureClassToFeatureClass("gpLayer", env.scratchGDB, "PMP_Points")
# save feature layer as "PMP_Points" feature class
        arcpy.AddMessage("(" + str(dm.GetCount("gpLayer")) + " grid points will be analyzed)\n")

        # Add PMP Fields
        for dur in durList:
            arcpy.AddMessage("\t...adding field: PMP_" + str(dur))
            dm.AddField(env.scratchGDB + "\\PMP_Points", "PMP_" + dur, "DOUBLE")

        # Add STORM Fields (this string values identifies the driving storm by SPAS ID number)
        for dur in durList:
            arcpy.AddMessage("\t...adding field: STORM_" + str(dur))
            dm.AddField(env.scratchGDB + "\\PMP_Points", "STORM_" + dur, "TEXT", "", "", 16,
"Storm ID " + dur + "-hour")

        # Add STNAME Fields (this string values identifies the driving storm by SPAS ID number)
#         for dur in durList:
#             arcpy.AddMessage("\t...adding field: STNAME_" + str(dur))
#             dm.AddField(env.scratchGDB + "\\PMP_Points", "STNAME_" + dur, "TEXT", "", "",
50, "Storm Name " + dur + "-hour")

        return

    ##########################################################################
    ##  Define getAOIarea() function:
    ##  getAOIarea() calculates the area of AOI (basin outline) input shapefile/
    ##  featureclass.  The basin outline shapefile must be projected.  The area
    ##  is sqaure miles, converted from the basin layers projected units (feet
    ##  or meters).  The aoiBasin feature class should only have a single feature
    ##  (the basin outline).  If there are multiple features, the area will be stored
    ##  for the final feature only.

    def getAOIarea():
        sr = arcpy.Describe(aoiBasin).SpatialReference                    # Determine
aoiBasin spatial reference system
        srname = sr.name
        srtype = sr.type
```

```
    srunitname = sr.linearUnitName                                    # Units
    arcpy.AddMessage("\nAOI basin spatial reference:  " + srname + "\nUnit type: " +
srunitname + "\nSpatial reference type: " + srtype)

    aoiArea = 0.0
    rows = arcpy.SearchCursor(aoiBasin)
    for row in rows:
        feat = row.getValue("Shape")
        aoiArea += feat.area
    if srtype == 'Geographic':                          # Must have a surface projection.  If one
doesn't exist it projects a temporary file and uses that.
        arcpy.AddMessage("\n***The basin shapefile's spatial reference 'Geographic' is not
supported.  Projecting temporary shapefile for AOI.***")
        arcpy.Project_management(aoiBasin,env.scratchGDB + "\\TempBasin",102039)     #
Projects AOI Basin (102039 = USA_Contiguous_Albers_Equal_Area_Conic_USGS_version)
        TempBasin = env.scratchGDB + "\\TempBasin"                                   # Path to
temporary basin created in scratch geodatabase
        sr = arcpy.Describe(TempBasin).SpatialReference                              # Determine
Spatial Reference of temporary basin
        aoiArea = 0.0
        rows = arcpy.SearchCursor(TempBasin)                                         #
Assign area size in square meters
        for row in rows:
            feat = row.getValue("Shape")
            aoiArea += feat.area
        aoiArea = aoiArea * 0.000000386102                                  # Converts square
meters to square miles
    elif srtype == 'Projected':
        if srunitname == "Meter":
            aoiArea = aoiArea * 0.000000386102                               # Converts square
meters to square miles
        elif srunitname == "Foot" or "Foot_US":
            aoiArea = aoiArea * 0.00000003587                                # Converts square
feet to square miles
        else:
            arcpy.AddMessage("\nThe basin shapefile's unit type '" + srunitname + "' is not
supported.")
            sys.exit("Invalid linear units")                                 # Units must be meters or
feet

    aoiArea = round(aoiArea, 3)
    arcpy.AddMessage("\nArea of interest: " + str(aoiArea) + " square miles.")

    if arcpy.GetParameter(6) == False:
        aoiArea = arcpy.GetParameter(7)                                      # Enable a constant
area size
```

```
    aoiArea = round(aoiArea, 1)
    arcpy.AddMessage("\n***Area used for PMP analysis: " + str(aoiArea) + " sqmi***")
    return aoiArea

##########################################################################
## Define dadLookup() function:
## The dadLookup() function determines the DAD value for the current storm
## and duration according to the basin area size. The DAD depth is interpolated
## linearly between the two nearest areal values within the DAD table.
def dadLookup(stormLayer, duration, area):          # dadLookup() accepts the current
storm layer name (string), the current duration (string), and AOI area size (float)
    #arcpy.AddMessage("\t\tfunction dadLookup() called.")
    durField = "H_" + duration                      # defines the name of the duration field (eg.,
"H_06" for 6-hour)
    dadTable = dadGDB + "\\" + stormLayer
    rows = arcpy.SearchCursor(dadTable)

    try:
      row = rows.next()                             # Sets DAD area x1 to the value in the first
row of the DAD table.
      x1 = row.AREASQMI
      y1 = row.getValue(durField)
      xFlag = "FALSE"                               # xFlag will remain false for basins that are
larger than the largest DAD area.
    except RuntimeError:                            # return if duration does not exist in DAD
table
      return

    row = rows.next()
    i = 0
    while row:                                      # iterates through the DAD table - assiging the
bounding values directly above and below the basin area size
      i += 1
      if row.AREASQMI < area:
        x1 = row.AREASQMI
        y1 = row.getValue(durField)
      else:
        xFlag = "TRUE"                              # xFlag is switched to "TRUE" indicating area
is within DAD range
        x2 = row.AREASQMI
        y2 = row.getValue(durField)
        break

      row = rows.next()
    del row, rows, i
```

```
        if xFlag == "FALSE":
            x2 = area                                   # If x2 is equal to the basin area, this means that the
largest DAD area is smaller than the basin and the resulting DAD value must be extrapolated.
            arcpy.AddMessage("\t\tThe basin area size: " + str(area) + " sqmi is greater than the
largest DAD area: " + str(x1) + " sqmi.\n\t\tDAD value is estimated by extrapolation.")
            y = x1 / x2 * y1                            # y (the DAD depth) is estimated by extrapolating
the DAD area to the basin area size.
            return y                                    # The extrapolated DAD depth (in inches) is returned.

        # arcpy.AddMessage("\nArea = " + str(area) + "\nx1 = " + str(x1) + "\nx2 = " + str(x2) +
"\ny1 = " + str(y1) + "\ny2 = " + str(y2))

        x = area                                        # If the basin area size is within the DAD table area
range, the DAD depth is interpolated
        deltax = x2 - x1                                # to determine the DAD value (y) at area (x) based
on next lower (x1) and next higher (x2) areas.
        deltay = y2 - y1
        diffx = x - x1

        y = y1 + diffx * deltay / deltax

        if x < x1:
            arcpy.AddMessage("\t\tThe basin area size: " + str(area) + " sqmi is less than the smallest
DAD table area: " + str(x1) + " sqmi.\n\t\tDAD value is estimated by extrapolation.")

        return y                                        # The interpolated DAD depth (in inches) is returned.

    ##########################################################################
    ##  Define updatePMP() function:
    ##  This function updates the 'PMP_XX_' and 'STORM_XX' fields of the PMP_Points
    ##  feature class with the largest value from all analyzed storms stored in the
    ##  pmpValues list.
    def updatePMP(pmpValues, stormID, duration):                                    # Accepts
four arguments: pmpValues - largest adjusted rainfall for current duration (float list); stormID -
driver storm ID for each PMP value (text list); and duration (string)
        pmpfield = "PMP_" + duration
        stormfield = "STORM_" + duration
        stormTextField = "STNAME_" + duration

        gridRows = env.scratchGDB + "\\PMP_Points"                                  # iterates
through PMP_Points rows
        i = 0
        with arcpy.da.UpdateCursor(gridRows, (pmpfield, stormfield)) as cursor:
            for row in cursor:
                row[0] = pmpValues[i]                                               # Sets the PMP field
value equal to the Max Adj. Rainfall value (if larger than existing value).
```

```
            row[1] = stormID[i]                                    # Sets the storm ID field
to indicate the driving storm event
            cursor.updateRow(row)
            i += 1
        del row, gridRows, pmpfield, stormfield, i
        arcpy.AddMessage("\n\t" + duration + "-hour PMP values update complete. \n")
        return

    ##########################################################################
    ##  The outputPMP() function produces raster GRID files for each of the PMP durations.
    ##  Aslo, a space-delimited PMP_Distribition.txt file is created in the 'Text_Output' folder.
    def outputPMP(type, area, outPath):
        desc = arcpy.Describe(basin)
        basinName = desc.baseName
        pmpPoints = env.scratchGDB + "\\PMP_Points"                    # Location of
'PMP_Points' feature class which will provide data for output

        outType = type[:1]
        outArea = str(int(round(area,0))) + "sqmi"
        outGDB = "PMP_"+ basinName + "_" + outArea +".gdb"
        if not arcpy.Exists(outPath + "\\" + outGDB):                  # Check to see if
PMP_XXXXX.gdb already exists
            arcpy.AddMessage("\nCreating output geodatabase '" + outGDB + "'")
            dm.CreateFileGDB(outPath, outGDB)
        arcpy.AddMessage("\nCopying PMP_Points feature class to " + outGDB + "...")
        con.FeatureClassToFeatureClass(pmpPoints, outPath + "\\" + outGDB, type +
"_PMP_Points_" + basinName + "_" + outArea)
        pointFC = outPath + "\\" + outGDB + "\\" + type + "_PMP_Points_" + basinName + "_" +
outArea
        # addLayerMXD(pointFC)  # calls addLayerMDX function to add output to ArcMap
session

        arcpy.AddMessage("\nBeginning PMP Raster Creation...")

        for dur in durList:                                            # This code creates a raster GRID from
the current PMP point layer
            durField = "PMP_" + dur
            outLoc = outPath + outGDB +"\\" + outType + "_" + dur + "_" + basinName + "_" +
outArea
            arcpy.AddMessage("\n\tInput Path: " + pmpPoints)
            arcpy.AddMessage("\tOutput raster path: " + outLoc)
            arcpy.AddMessage("\tField name: " + durField)
            con.FeatureToRaster(pmpPoints, durField, outLoc, "0.025")
            arcpy.AddMessage("\tOutput raster created...")
        del durField, outLoc, dur
```

```
    arcpy.AddMessage("\nPMP Raster Creation complete.")

    if includeSubbasin:                                  # Begin subbasin average calculations
        subbasinID = []
        with arcpy.da.SearchCursor(basin, subbasinIDfield) as cursor:   # Create list of subbasin
ID names
            for row in cursor:
                subbasinID.append(row[0])

        subIDtype = arcpy.ListFields(basin, subbasinIDfield)[0].type    # Define the datatype of
the subbasin ID field

        if subIDtype != "String":                        # Convert subbasin IDs to a string, if
they are not already
            subbasinID = [str(i) for i in subbasinID]

        subNameLen = max(map(len, subbasinID))                  # Define the length of the
longest subbasin ID

        # arcpy.AddMessage("\nList of subbasins...\n" + "\n".join(subbasinID))

        arcpy.AddMessage("\nCreating Subbasin Summary Table...")
        tableName = type + "_PMP_Subbasin_Average" + "_" + outArea
        tablePath = outPath + "\\" + outGDB + "\\" + tableName
        dm.CreateTable(outPath + "\\" + outGDB, tableName)          # Create blank table

        dm.AddField(tablePath, "STORM_TYPE", "TEXT", "", "", 10, "Storm Type")        #
Create "Storm Type" field
        dm.AddField(tablePath, "SUBBASIN", "TEXT", "", "", subNameLen, "Subbasin")     #
Create "Subbasin" field

        cursor = arcpy.da.InsertCursor(tablePath, "SUBBASIN")          # Create Insert cursor and
add a blank row to the table for each subbasin
        for sub in subbasinID:
            cursor.insertRow([sub])
        del cursor, sub

        dm.CalculateField(tablePath, "STORM_TYPE", "'" + type + "'", "PYTHON_9.3")     #
populate storm type field

        i = 0
        for field in arcpy.ListFields(pmpPoints, "PMP_*"):          # Add fields for each PMP
duration and calculate the subbasin averages
            fieldName = field.name
            arcpy.AddMessage("\n\tCalculating subbasin average for " + fieldName + "
(weighted)...\n")
```

```
            dm.AddField(tablePath, fieldName, "DOUBLE", "", 2)         # Add duration field
            subAveList = []
            for subbasin in subbasinID:                                # Loop through each subbasin
               if subIDtype != "String":                               # Define an SQL expression that
specifies the current subbasin
                  sql_exp = """{0} = {1}""".format(arcpy.AddFieldDelimiters(basin,
subbasinIDfield), subbasin)
               else:
                  sql_exp = """{0} = '{1}'""".format(arcpy.AddFieldDelimiters(basin,
subbasinIDfield), subbasin)
               dm.MakeFeatureLayer(basin, "subbasinLayer", sql_exp)
               outLayer = outPath + "\\" + outGDB + "\\subbasin_" + str(subbasin)
               subBasAve = basinAve("subbasinLayer", fieldName)        # Call the basAve()
function passing the subbasin and duration field
               arcpy.AddMessage("\tSubbasin average for " + str(subbasin) + ":  " +
str(subBasAve) + "")
               subAveList.append(subBasAve)                            # Add subbasin average to list
            p = 0
            with arcpy.da.UpdateCursor(tablePath, fieldName) as cursor: # Update the subbasin
average summary table with the subbasin averages
               for row in cursor:
                  row = subAveList[p]
                  cursor.updateRow([row])
                  p += 1

   ##          dm.CalculateField(tablePath, fieldName, fieldAve, "PYTHON_9.3")      # Assigns
the basin average
   ##          dur = durList[i]                                        # following lines add alias field
names to basin average table (ArcGIS 10.2.1 or later)
   ##          if dur[0] == "0":
   ##             dur = dur[1:]
   ##          fieldAlias = dur + "-hour PMP"
   ##          dm.AlterField(tablePath, fieldName, "#", fieldAlias)
            i += 1
         arcpy.AddMessage("\nSubbasin summary table complete.")

      arcpy.AddMessage("\nCreating Basin Summary Table...")
      tableName = type + "_PMP_Basin_Average" + "_" + outArea
      tablePath = outPath + "\\" + outGDB + "\\" + tableName
      dm.CreateTable(outPath + "\\" + outGDB, tableName)         # Create blank table
      cursor = arcpy.da.InsertCursor(tablePath, "*")             # Create Insert cursor and add a
blank row to the table
      cursor.insertRow([0])
      del cursor
```

```
    dm.AddField(tablePath, "STORM_TYPE", "TEXT", "", "", 30, "Storm Type")         # Create "Storm Type" field
    dm.CalculateField(tablePath, "STORM_TYPE", "'" + type + "'", "PYTHON_9.3")     # populate storm type field

    i = 0
    for field in arcpy.ListFields(pmpPoints, "PMP_*"):         # Add fields for each PMP duration and calculate the basin average
        fieldName = field.name
        fieldAve = basinAve(basin, fieldName)                  # Calls the basinAve() function - returns the average (weighted or not)
        dm.AddField(tablePath, fieldName, "DOUBLE", "", 2)     # Add duration field
        dm.CalculateField(tablePath, fieldName, fieldAve, "PYTHON_9.3")       # Assigns the basin average
##          dur = durList[i]                                   # following lines add alias field names to basin average table (ArcGIS 10.2.1 or later)
##          if dur[0] == "0":
##              dur = dur[1:]
##          fieldAlias = dur + "-hour PMP"
##          dm.AlterField(tablePath, fieldName, "#", fieldAlias)
        i += 1
    arcpy.AddMessage("\nSummary table complete.")
    basAveTables.append(tablePath)

##                  The following lines export a .png image depth duration chart and PMP summary excel file to the output folder
    if ddChart:
        xValues = durList                                      #Get list of durations for chart
        xValues = [int(i) for i in xValues]                    #Convert duration list to integers
        ax1 = plt.subplot2grid((1,1), (0,0))    #Create variable for subplot in chart
        yValues = []
        pmpFields = [field.name for field in arcpy.ListFields(tablePath, "PMP_*")] # Selects PMP fields for yValues
        with arcpy.da.SearchCursor(tablePath, pmpFields) as cursor:              # Adds PMP depths to yValues
            yValues = next(cursor)
        del cursor, pmpFields

        stormFields = [field.name for field in arcpy.ListFields(pmpPoints, "Storm_*")] # Selects Controlling Storm fields
        contStorms = []                        # List of controlling storms for a single duration
        listOfContStorms = []                  # List of controlling storms for all durations (list of lists)
        i = 0                                  # iterator (for "Storm_*" fields)
        while i < len(stormFields):                    # iterates through controlling storm fields
```

```
            with arcpy.da.SearchCursor(pmpPoints, stormFields) as cursor:    # Search cursor
returns list of unique controlling storms
                contStorms = sorted({row[i] for row in cursor})
            listOfContStorms.append(contStorms)                    # Add unique storms for current
duration to list of controlling stomrs
            i += 1
        del cursor

        plt.plot(xValues,yValues)               #Creates chart
        plt.xlabel('Storm Duration in Hours')
        plt.ylabel('Rainfall Depth in Inches')
        plt.title(basinName + " (" + outArea + ") " + type + ' Storm Basin Average PMP\nDepth
Duration Chart')
        ax1.grid(True)                          #Creates grid lines in chart
        yTop = max(yValues) + 1
        ax1.set_ylim(top = yTop)                #Sets y axis values to match depths +1 1
        ax1.set_xticks(xValues)                 #Sets x axis values to match durations
##          i = 0
##          xy = zip(xValues, yValues)
##          while i < len(stormFields):                 # iterates through controlling storm fields
##              pointXY = xy[i]
##              yLabel = '{0:.1f}'.format(yValues[i])       # round PMP depth to 1 decimal and
convert to string
##              stormLabel = str(listOfContStorms[i])       # convert controlling storm ID(s) to string
##              stormLabel = stormLabel.replace("u", "")    # remove unicode "u"
##              stormLabel = stormLabel.replace("'", "")    # remove unicode ","
##              stormLabel = stormLabel.replace("[", "")    # remove unicode "["
##              stormLabel = stormLabel.replace("]", "")    # remove unicode "]"
##              #ax1.annotate(yLabel + '"\n' + stormLabel, xy=xy[i], textcoords='offset points',
size=8, annotation_clip=True)
##              ax1.annotate(yLabel + '"\n' + stormLabel, xy=xy[i], textcoords='data', size=8,
annotation_clip=True)
##              i += 1
##          del xy

        plt.savefig(outPath + "\\" + basinName + "_" + type + "_Depth_Duration_Chart.png")
        #Save image
        plt.close()                         #Close chart to remove from memory
        arcpy.AddMessage("\nDepth Duration Chart exported to output folder.")
        del xValues, yValues, #df, dfLimited
        return
    return

#################################################################################
##  The basin() returns the basin average PMP value for a given duration field.
##  If the option for a weighted average is checked in the tool parameter the script
```

```
   ##  will weight the grid point values based on proportion of area inside the basin.
   def basinAve(aoiBasin, pmpField):
      pmpPoints = env.scratchGDB + "\\PMP_Points"                                    # Path
of 'PMP_Points' scratch feature class
      if weightedAve:
         #arcpy.AddMessage("\tCalculating sub-basin average for " + pmpField + "(weighted)...")
         vectorGridClip = env.scratchGDB + "\\VectorGridClip"                        # Path
of 'VectorGridClip' scratch feature class

         dm.MakeFeatureLayer(home + "\\Input\\Non_Storm_Data.gdb\\Vector_Grid",
"vgLayer")              # make a feature layer of vector grid cells
         dm.SelectLayerByLocation("vgLayer", "INTERSECT", aoiBasin)
# select the vector grid cells that intersect the aoiBasin polygon

         an.Clip("vgLayer", aoiBasin, vectorGridClip)                                 #
clips aoi vector grid to basin
         dm.AddField(pmpPoints, "WEIGHT", "DOUBLE")                                  #
adds 'WEIGHT' field to PMP_Points scratch feature class
         dm.MakeFeatureLayer(vectorGridClip, "vgClipLayer")                          #
make a feature layer of basin clipped vector grid cells
         dm.MakeFeatureLayer(pmpPoints, "pmpPointsLayer")                            #
make a feature layer of PMP_Points feature class

         dm.AddJoin("pmpPointsLayer", "ID", "vgClipLayer", "ID")
# joins PMP_Points and vectorGridBasin tables
         dm.CalculateField("pmpPointsLayer", "WEIGHT", "!vectorGridClip.Shape_Area!",
"PYTHON_9.3")          # Calculates basin area proportion to use as weight for each grid cell.
         dm.RemoveJoin("pmpPointsLayer", "vectorGridClip")

         dm.SelectLayerByLocation("pmpPointsLayer", "INTERSECT", "vgLayer")

         na = arcpy.da.TableToNumPyArray("pmpPointsLayer",(pmpField, 'WEIGHT'))
# Assign pmpPoints values and weights to Numpy array (na)
         wgtAve = np.average(na[pmpField], weights=na['WEIGHT'])                     #
Calculate weighted average with Numpy average
         del na
         return round(wgtAve, 2)

      else:
         if includeSubbasin:
            #arcpy.AddMessage("\tCalculating sub-basin average for " + pmpField + "(non-
weighted)...")
            vectorGridClip = env.scratchGDB + "\\VectorGridClip"                     #
Path of 'VectorGridClip' scratch feature class
```

```
            dm.MakeFeatureLayer(home + "\\Input\\Non_Storm_Data.gdb\\Vector_Grid",
"vgLayer")              # make a feature layer of vector grid cells
            dm.SelectLayerByLocation("vgLayer", "INTERSECT", aoiBasin)
# select the vector grid cells that intersect the aoiBasin polygon

            dm.MakeFeatureLayer(pmpPoints, "pmpPointsLayer")                                    #
make a feature layer of PMP_Points feature class

            dm.SelectLayerByLocation("pmpPointsLayer", "INTERSECT", "vgLayer")

            na = arcpy.da.TableToNumPyArray("pmpPointsLayer", pmpField)
# Assign pmpPoints values and weights to Numpy array (na)
            fieldAve = np.average(na[pmpField])                                                 #
Calculates aritmetic mean
            del na
            return round(fieldAve, 2)

         else:
            arcpy.AddMessage("\tCalculating basin average for " + pmpField + "(not
weighted)...")
            na = arcpy.da.TableToNumPyArray(pmpPoints, pmpField)
# Assign pmpPoints values to Numpy array (na)
            fieldAve = np.average(na[pmpField])                                                 #
Calculates aritmetic mean
            del na
            return round(fieldAve, 2)

   ###############################################################################
   ##  This basinZone() function returns a list containing transposition zone ID
   ##  (as an integer)


   def   basinZone(bas):          ## This function returns the basin location transposition zone
      tempBasin = env.scratchGDB + "\\tempBasin"
      tempCentroid = env.scratchGDB + "\\tempCentroid"
      joinFeat = home + "\\Input\\Non_Storm_Data.gdb\\Vector_Grid"
      joinOutput = env.scratchGDB + "\\joinOut"
      dm.Dissolve(bas, tempBasin)
      desc = arcpy.Describe(tempBasin)
      sr = desc.spatialReference
      #dm.FeatureToPoint(tempBasin, tempCentroid, "INSIDE")

      dm.CreateFeatureclass(env.scratchGDB,"tempCentroid","POINT",spatial_reference = sr)
      with arcpy.da.InsertCursor(tempCentroid, "SHAPE@XY") as iCur:
         with arcpy.da.SearchCursor(tempBasin,"SHAPE@") as sCur:
            for sRow in sCur:
```

```
            cent = sRow[0].centroid          # get the centroid
            iCur.insertRow([(cent.X,cent.Y)])# write it to the new feature class

        an.SpatialJoin(tempCentroid, joinFeat, joinOutput)
        centZone = arcpy.da.SearchCursor(joinOutput, ("ZONE",)).next()[0]
        del tempBasin, tempCentroid, joinFeat, joinOutput, desc, sr
        return (centZone)

    ###############################################################################
    ##  The temporalDist() functions applies the temporal distributions scenarios
    ##  to PMP.


    def temporalDistControlStorm_06hr(stormType, outPath, location, areaSize, basinName):
# Local Storm 6-hr Temporal Distributions Function

        basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
        basinPMPPoints = outPath + "\\" + stormType + "_PMP_Points_" + basinName + "_" +
areaSize                        # Location of basin average PMP table
        controlStormTable = home +
"\\Input\\Non_Storm_Data.gdb\\CONTROLLING_STORM_TEMPORAL_DISTRIBUTIONS_0
6"
        arcpy.AddMessage(stormType + " Storm - " + dur + "-hour Controlling Storm PMP
Temporal Distributions***")
        outTable = outPath + "\\Controlling_Storms_Temporal_Distributions_" + dur
        pointsArray = arcpy.da.TableToNumPyArray(basinPMPPoints, "Storm_" + dur)
        arrayList = []
        for r in pointsArray:
            arrayList.append(r[0])
        distributionList = np.unique(arrayList).tolist()
        controlPatterns = [f.name for f in arcpy.ListFields(controlStormTable)]
        TF = any(item in distributionList for item in controlPatterns)
        if TF == True:
            map = arcpy.FieldMappings()
            fm = arcpy.FieldMap()
            fm.addInputField(controlStormTable, "TIMESTEP")
            map.addFieldMap(fm)
            fm2 = arcpy.FieldMap()
            fm2.addInputField(controlStormTable, "MINUTE")
            map.addFieldMap(fm2)
            for field in distributionList:
                if (field in controlPatterns):
                    fm3 = arcpy.FieldMap()
                    fm3.addInputField(controlStormTable, field)
                    map.addFieldMap(fm3)
            arcpy.AddMessage("\n\tCreating temporal distribution table:...")
```

```
        arcpy.TableToTable_conversion(controlStormTable, outPath,
"Controlling_Storms_Temporal_Distributions_" + dur, "", map)                    # Copy 6-
hour temporal dist. factors table to output location
        sixHour = arcpy.da.SearchCursor(basinPMP, ("PMP_06",)).next()[0]          # Gets
6-hour PMP depth
        for distribution in distributionList:                                  # Loops thourgh each 6-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            with arcpy.da.UpdateCursor(outTable, distribution) as cursor:          # Cursor to
apply temporal factor to 6-hour PMP
                for row in cursor:
                    row[0] = row[0] * sixHour
                    cursor.updateRow(row)
                del row, cursor

        dists6hr = []       # add suffix to distribution pattern name
        for dist in distributionList:
            dists6hr.append(dist + " (6-hr)")

        checkTemporal(stormType, outPath, outTable, dists6hr, dur, areaSize)
    else:
        arcpy.AddMessage("***Controlling Storm does not have any temporal distributions for
this duration***")

    return


  def temporalDistControlStorm_12hr(stormType, outPath, location, areaSize, basinName):
# Local Storm 12-hr Temporal Distributions Function

    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
    basinPMPPoints = outPath + "\\" + stormType + "_PMP_Points_" + basinName + "_" +
areaSize                    # Location of basin average PMP table
    controlStormTable = home +
"\\Input\\Non_Storm_Data.gdb\\CONTROLLING_STORM_TEMPORAL_DISTRIBUTIONS_1
2"
    arcpy.AddMessage(stormType + " Storm - " + dur + "-hour Controlling Storm PMP
Temporal Distributions***")
    outTable = outPath + "\\Controlling_Storms_Temporal_Distributions_" + dur
    pointsArray = arcpy.da.TableToNumPyArray(basinPMPPoints, "Storm_" + dur)
    arrayList = []
    for r in pointsArray:
        arrayList.append(r[0])
    distributionList = np.unique(arrayList).tolist()
    controlPatterns = [f.name for f in arcpy.ListFields(controlStormTable)]
    TF = any(item in distributionList for item in controlPatterns)
```

```
        if TF == True:
            map = arcpy.FieldMappings()
            fm = arcpy.FieldMap()
            fm.addInputField(controlStormTable, "TIMESTEP")
            map.addFieldMap(fm)
            fm2 = arcpy.FieldMap()
            fm2.addInputField(controlStormTable, "MINUTE")
            map.addFieldMap(fm2)
            for field in distributionList:
                fm3 = arcpy.FieldMap()
                fm3.addInputField(controlStormTable, field)
                map.addFieldMap(fm3)
            arcpy.AddMessage("\n\tCreating temporal distribution table:...")
            arcpy.TableToTable_conversion(controlStormTable, outPath,
"Controlling_Storms_Temporal_Distributions_" + dur, "", map)                    # Copy 12-
hour temporal dist. factors table to output location
            twelveHour = arcpy.da.SearchCursor(basinPMP, ("PMP_12",)).next()[0]          #
Gets 12-hour PMP depth
            for distribution in distributionList:                    # Loops thourgh each 12-hour
temporal distribution
                arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
                with arcpy.da.UpdateCursor(outTable, distribution) as cursor:          # Cursor to
apply temporal factor to 12-hour PMP
                    for row in cursor:
                        row[0] = row[0] * twelveHour
                        cursor.updateRow(row)
                    del row, cursor

            dists12hr = []      # add suffix to distribution pattern name
            for dist in distributionList:
                dists12hr.append(dist + " (12-hr)")

            checkTemporal(stormType, outPath, outTable, dists12hr, dur, areaSize)
        else:
            arcpy.AddMessage("***Controlling Storm does not have any temporal distributions for
this duration***")

        return


    def temporalDistControlStorm_24hr(stormType, outPath, location, areaSize, basinName):
# 24-hr Temporal Distributions Function
        basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
        basinPMPPoints = outPath + "\\" + stormType + "_PMP_Points_" + basinName + "_" +
areaSize                    # Location of basin average PMP table
```

```
        controlStormTable = home +
"\\Input\\Non_Storm_Data.gdb\\CONTROLLING_STORM_TEMPORAL_DISTRIBUTIONS_2
4"

        arcpy.AddMessage(stormType + " Storm - " + dur + "-hour Controlling Storm PMP
Temporal Distributions***")
        outTable = outPath + "\\Controlling_Storms_Temporal_Distributions_" + dur
        pointsArray = arcpy.da.TableToNumPyArray(basinPMPPoints, "Storm_" + dur)
        arrayList = []
        for r in pointsArray:
            arrayList.append(r[0])
        distributionList = np.unique(arrayList).tolist()
        controlPatterns = [f.name for f in arcpy.ListFields(controlStormTable)]
        TF = any(item in distributionList for item in controlPatterns)
        if TF == True:
            map = arcpy.FieldMappings()
            fm = arcpy.FieldMap()
            fm.addInputField(controlStormTable, "TIMESTEP")
            map.addFieldMap(fm)
            fm2 = arcpy.FieldMap()
            fm2.addInputField(controlStormTable, "MINUTE")
            map.addFieldMap(fm2)
            for field in distributionList:
                fm3 = arcpy.FieldMap()
                fm3.addInputField(controlStormTable, field)
                map.addFieldMap(fm3)
            arcpy.AddMessage("\n\tCreating temporal distribution table:...")
            arcpy.TableToTable_conversion(controlStormTable, outPath,
"Controlling_Storms_Temporal_Distributions_" + dur, "", map)                    # Copy 24-
hour temporal dist. factors table to output location
            twentyfourHour = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]
# Gets 24-hour PMP depth
            for distribution in distributionList:                              # Loops thourgh each 24-hour
temporal distribution
                arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
                with arcpy.da.UpdateCursor(outTable, distribution) as cursor:          # Cursor to
apply temporal factor to 24-hour PMP
                    for row in cursor:
                        row[0] = row[0] * twentyfourHour
                        cursor.updateRow(row)
                    del row, cursor

            dists24hr = []          # add suffix to distribution pattern name
            for dist in distributionList:
                dists24hr.append(dist + " (24-hr)")
```

```
        checkTemporal(stormType, outPath, outTable, dists24hr, dur, areaSize)
    else:
        arcpy.AddMessage("***Controlling Storm does not have any temporal distributions for
this duration***")

    return


  def temporalDistControlStorm_72hr(stormType, outPath, location, areaSize, basinName):
# Local Storm 72-hr Temporal Distributions Function

    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
    basinPMPPoints = outPath + "\\" + stormType + "_PMP_Points_" + basinName + "_" +
areaSize                    # Location of basin average PMP table
    controlStormTable = home +
"\\Input\\Non_Storm_Data.gdb\\CONTROLLING_STORM_TEMPORAL_DISTRIBUTIONS_7
2"
    arcpy.AddMessage(stormType + " Storm - " + dur + "-hour Controlling Storm PMP
Temporal Distributions***")
    outTable = outPath + "\\Controlling_Storms_Temporal_Distributions_" + dur
    pointsArray = arcpy.da.TableToNumPyArray(basinPMPPoints, "Storm_" + dur)
    arrayList = []
    for r in pointsArray:
        arrayList.append(r[0])
    distributionList = np.unique(arrayList).tolist()
    controlPatterns = [f.name for f in arcpy.ListFields(controlStormTable)]
    TF = any(item in distributionList for item in controlPatterns)
    if TF == True:
        map = arcpy.FieldMappings()
        fm = arcpy.FieldMap()
        fm.addInputField(controlStormTable, "TIMESTEP")
        map.addFieldMap(fm)
        fm2 = arcpy.FieldMap()
        fm2.addInputField(controlStormTable, "MINUTE")
        map.addFieldMap(fm2)
        for field in distributionList:
            fm3 = arcpy.FieldMap()
            fm3.addInputField(controlStormTable, field)
            map.addFieldMap(fm3)
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        arcpy.TableToTable_conversion(controlStormTable, outPath,
"Controlling_Storms_Temporal_Distributions_" + dur, "", map)                    # Copy 72-
hour temporal dist. factors table to output location
        seventyTwoHour = arcpy.da.SearchCursor(basinPMP, ("PMP_72",)).next()[0]
# Gets 72-hour PMP depth
```

```
        for distribution in distributionList:                    # Loops thourgh each 72-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            with arcpy.da.UpdateCursor(outTable, distribution) as cursor:          # Cursor to
apply temporal factor to 72-hour PMP
                for row in cursor:
                    row[0] = row[0] * seventyTwoHour
                    cursor.updateRow(row)
                del row, cursor

        dists72hr = []      # add suffix to distribution pattern name
        for dist in distributionList:
            dists72hr.append(dist + " (72-hr)")

        checkTemporal(stormType, outPath, outTable, dists72hr, dur, areaSize)
    else:
        arcpy.AddMessage("***Controlling Storm does not have any temporal distributions for
this duration***")

    return


  def temporalDistLS2(stormType, outPath, location, areaSize):          # Local Storm 2-
hr Temporal Distribution Function
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table

    if stormType == "Local":
        arcpy.AddMessage("\n***Local Storm - 2-hour PMP Temporal Distribution***")

        temporalDistTable_2hr = home +
"\\Input\\Non_Storm_Data.gdb\\LS_TEMPORAL_DISTRIBUTIONS_02HR"  # 2-hour
Temporal distribution factors table
        outTable = outPath + "\\LS_Temporal_Distributions_02hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_2hr, outTable)                  # Copy 2-hour temporal
dist. factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_2hr,
"LS*")]    # Create a list of 2-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))

        oneHour = arcpy.da.SearchCursor(basinPMP, ("PMP_01",)).next()[0]          # Gets
1-hour PMP depth
        twoHour = arcpy.da.SearchCursor(basinPMP, ("PMP_02",)).next()[0]          # Gets
2-hour PMP depth
```

```
        for distribution in distributionList:                    # Loops through each 2-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            accumPMP = 0
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to apply temporal factor to 2-hour PMP
                for row in cursor:
                    if row[1] <= 6:                              # Leave loop once a specified row is
reached
                        accumPMP +=  (twoHour - oneHour) / 12
                        row[0] = accumPMP
                        cursor.updateRow(row)
                    if row[1] > 6 and row[1] <= 18:              # Constrain update to rows 7-
18
                        accumPMP += oneHour * row[0]
                        row[0] = accumPMP
                        cursor.updateRow(row)
                    if row[1] > 18 and row[1] <= 24:             # Constrain update to rows
19-24
                        accumPMP +=  (twoHour - oneHour) / 12
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)  # Calls
checkTemporal function
    return


  def temporalDistLS(stormType, outPath, location, areaSize):          # Local Storm 6-
hr Temporal Distributions Function
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table

    if stormType == "Local":
        arcpy.AddMessage("\n***Local Storm - 6-hour PMP Temporal Distributions***")

        temporalDistTable_6hr = home +
"\\Input\\Non_Storm_Data.gdb\\LS_TEMPORAL_DISTRIBUTIONS_06HR"  # 6-hour
Temporal distribution factors table
        outTable = outPath + "\\LS_Temporal_Distributions_6hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_6hr, outTable)                    # Copy 6-hour temporal
dist. factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_6hr,
"LS*")]    # Create a list of 6-hour distribution field names
```

```
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))
        sixHour = arcpy.da.SearchCursor(basinPMP, ("PMP_06",)).next()[0]                # Gets
6-hour PMP depth
        for distribution in distributionList:                                  # Loops thourgh each 6-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            with arcpy.da.UpdateCursor(outTable, distribution) as cursor:              # Cursor to
apply temporal factor to 6-hour PMP
                for row in cursor:
                    row[0] = row[0] * sixHour
                    cursor.updateRow(row)
                del row, cursor
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)
    return


  def temporalDist_LS12hr(stormType, outPath, location, areaSize):                    # Local
Storm 12-hr Temporal Distributions Function
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table

    if stormType == "Local":
        arcpy.AddMessage("\n***" + stormType + " Storm - 12hr PMP Temporal
Distributions***")

        temporalDistTable_12hr = home +
"\\Input\\Non_Storm_Data.gdb\\LS_TEMPORAL_DISTRIBUTIONS_12HR"    # Local Storm
Temporal distribution factors table
        outTable = outPath + "\\LS_Temporal_Distributions_12hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_12hr, outTable)                          # Copy temporal dist.
factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_12hr,
"LS*")]       # Create a list of 12-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))

        largest6 = arcpy.da.SearchCursor(basinPMP, ("PMP_06",)).next()[0]                #
Calculate largest 6-hour period PMP
        second6 = (arcpy.da.SearchCursor(basinPMP, ("PMP_12",)).next()[0] - largest6)/2
# Calculate the next largest 6-hr period PMP and divide by 2

        arcpy.AddMessage("\n\tLargest 6-hour Period: " + str(largest6))
        arcpy.AddMessage("\tFirst 3-hour: " + str(second6))
        arcpy.AddMessage("\tLast 3-hour: " + str(second6))
```

```
        for distribution in distributionList:                    # Loops thourgh each 12-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            arcpy.AddMessage("\t\tFirst 3-hour Period...")
            accumPMP = 0
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to evenly distribute half of 2nd largest 6-hr into first 3 hours
                for row in cursor:
                    if row[1] <= 36:                             # Leave loop once a row containing a
temporal dist. factor (ie, first 3h period) is reached
                        accumPMP +=  second6 / 36
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tLargest 6-hour Period...")
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to apply temporal factors to largest 6-hour PMP
                for row in cursor:
                    if row[1] > 36 and row[1] <= 108:            # Constrain update to rows
36-108 (middle 6hr period)
                        accumPMP = (largest6 * row[0]) + second6
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tLast 3-hour Period...")
            whereClause = distribution + " IS NULL"
            with arcpy.da.UpdateCursor(outTable, distribution, whereClause) as cursor:          #
Cursor to evenly distribute half of 2nd largest 6-hr into last 3 hours
                for row in cursor:
                    accumPMP +=  second6 / 36
                    row[0] = accumPMP
                    cursor.updateRow(row)
                del row, cursor, accumPMP, whereClause
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)
    return


  def temporalDist_24hr(stormType, outPath, location, areaSize):                    #
General/Tropical Storm 24-hr Temporal Distributions Function
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table

    if stormType == "General":
```

```
        arcpy.AddMessage("\n***" + stormType + " Storm - 24hr PMP Temporal
Distributions***")

        temporalDistTable_24hr = home +
"\\Input\\Non_Storm_Data.gdb\\GS_TEMPORAL_DISTRIBUTIONS_24HR"    # General
Storm Temporal distribution factors table
        outTable = outPath + "\\GS_Temporal_Distributions_24hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_24hr, outTable)                    # Copy temporal dist.
factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_24hr,
"GS*")]       # Create a list of 24-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))
        twentyfourHour = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]              #
Gets 24-hour PMP depth
        for distribution in distributionList:                        # Loops thourgh each 24-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            with arcpy.da.UpdateCursor(outTable, distribution) as cursor:              # Cursor to
apply temporal factor to 24-hour PMP
                for row in cursor:
                    row[0] = row[0] * twentyfourHour
                    cursor.updateRow(row)
                del row, cursor
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)

    if stormType == "Tropical":
        arcpy.AddMessage("\n***" + stormType + " Storm - 24hr PMP Temporal
Distributions***")

        temporalDistTable_24hr = home +
"\\Input\\Non_Storm_Data.gdb\\TS_TEMPORAL_DISTRIBUTIONS_24HR"    # Tropical
Storm Temporal distribution factors table
        outTable = outPath + "\\TS_Temporal_Distributions_24hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_24hr, outTable)                    # Copy temporal dist.
factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_24hr,
"TS*")]       # Create a list of 24-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))
        twentyfourHour = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]              #
Gets 24-hour PMP depth
        for distribution in distributionList:                        # Loops thourgh each 24-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
```

```
            with arcpy.da.UpdateCursor(outTable, distribution) as cursor:                # Cursor to
apply temporal factor to 24-hour PMP
                for row in cursor:
                    row[0] = row[0] * twentyfourHour
                    cursor.updateRow(row)
                del row, cursor
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)
    return


  def temporalDist_48hr(stormType, outPath, location, areaSize):                    #
General/Tropical Storm 48-hr Temporal Distributions Function
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table

    if stormType == "General":
        arcpy.AddMessage("\n***" + stormType + " Storm - 48hr PMP Temporal
Distributions***")

        temporalDistTable_48hr = home +
"\\Input\\Non_Storm_Data.gdb\\GS_TEMPORAL_DISTRIBUTIONS_48HR"    # General
Storm Temporal distribution factors table
        outTable = outPath + "\\GS_Temporal_Distributions_48hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_48hr, outTable)                                # Copy temporal dist.
factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_48hr,
"GS*")]       # Create a list of 48-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))

        largest24 = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]                #
Calculate largest 24-hour period PMP
        second24 = (arcpy.da.SearchCursor(basinPMP, ("PMP_48",)).next()[0] - largest24)/2
# Calculate the next largest 24-hr period PMP and divide by 2

        arcpy.AddMessage("\n\tLargest 24-hour Period: " + str(largest24))
        arcpy.AddMessage("\tFirst 12-hour: " + str(second24))
        arcpy.AddMessage("\tLast 12-hour: " + str(second24))


        for distribution in distributionList:                                # Loops thourgh each 24-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            arcpy.AddMessage("\t\tFirst 12-hour Period...")
            accumPMP = 0
```

```
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to evenly distribute half of 2nd largest 24-hr into first 12 hours
                for row in cursor:
                    if row[1] <= 48:                                # Leave loop once a row containing a
temporal dist. factor (ie, first 12h period) is reached
                        accumPMP +=  second24 / 48
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tLargest 24-hour Period...")
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to apply temporal factors to largest 24-hour PMP
                for row in cursor:
                    if row[1] > 48 and row[1] <= 144:                # Constrain update to rows
49-144 (second 24hr period)
                        accumPMP = (largest24 * row[0]) + second24
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tLast 12-hour Period...")
            whereClause = distribution + " IS NULL"
            with arcpy.da.UpdateCursor(outTable, distribution, whereClause) as cursor:          #
Cursor to evenly distribute half of 2nd largest 24-hr into last 12 hours
                for row in cursor:
                    accumPMP +=  second24 / 48
                    row[0] = accumPMP
                    cursor.updateRow(row)
                del row, cursor, accumPMP, whereClause
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)

    if stormType == "Tropical":
        arcpy.AddMessage("\n***" + stormType + " Storm - 48hr PMP Temporal
Distributions***")

        temporalDistTable_48hr = home +
"\\Input\\Non_Storm_Data.gdb\\TS_TEMPORAL_DISTRIBUTIONS_48HR"    # Tropical
Storm Temporal distribution factors table
        outTable = outPath + "\\TS_Temporal_Distributions_48hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_48hr, outTable)                       # Copy temporal dist.
factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_48hr,
"TS*")]       # Create a list of 48-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))
```

```
        largest24 = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]                    #
Calculate largest 24-hour period PMP
        second24 = (arcpy.da.SearchCursor(basinPMP, ("PMP_48",)).next()[0] - largest24)/2
# Calculate the third largest 12-hr period PMP and divide by 2


        arcpy.AddMessage("\n\tLargest 24-hour Period: " + str(largest24))
        arcpy.AddMessage("\tFirst 12-hour: " + str(second24))
        arcpy.AddMessage("\tLast 12-hour: " + str(second24))


        for distribution in distributionList:                              # Loops thourgh each 24-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            arcpy.AddMessage("\t\tFirst 12-hour Period...")
            accumPMP = 0
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to evenly distribute half of 2nd largest 24-hour PMP to first 12 hours
                for row in cursor:
                    if row[1] <= 48:                                # Leave loop once a row containing a
temporal dist. factor (ie, first 12h period) is reached
                        accumPMP +=  second24 / 48
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tLargest 24-hour Period...")
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to apply temporal factors to largest 24-hour PMP
                for row in cursor:
                    if row[1] > 48 and row[1] <= 144:                        # Constrain update to rows
49-144 (second 24hr period)
                        accumPMP = (largest24 * row[0]) + second24
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tLast 12-hour Period...")
            whereClause = distribution + " IS NULL"
            with arcpy.da.UpdateCursor(outTable, distribution, whereClause) as cursor:             #
Cursor to evenly distribute half of 2nd largest 24-hr into last 12 hours
                for row in cursor:
                    accumPMP +=  second24 / 48
                    row[0] = accumPMP
                    cursor.updateRow(row)
```

```
            del row, cursor, accumPMP, whereClause

        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)
    return


  def temporalDist_72hr(stormType, outPath, location, areaSize):                    #
General/Tropical Storm 72-hr Temporal Distributions Function
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table

    if stormType == "General":
        arcpy.AddMessage("\n***" + stormType + " Storm - 72hr PMP Temporal
Distributions***")
        temporalDistTable_72hr = home +
"\\Input\\Non_Storm_Data.gdb\\GS_TEMPORAL_DISTRIBUTIONS_72HR"    # General
Storm Temporal distribution factors table
        outTable = outPath + "\\GS_Temporal_Distributions_72hr"
        arcpy.AddMessage("\n\tCreating temporal distribution table:...")
        dm.Copy(temporalDistTable_72hr, outTable)                              # Copy temporal dist.
factors table to output location
        distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_72hr,
"GS*")]       # Create a list of 72-hour distribution field names
        arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))

        largest24 = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]
# Calculate largest 24-hour period PMP
        second24 = arcpy.da.SearchCursor(basinPMP, ("PMP_48",)).next()[0] - largest24
# Calculate 2nd-largest 24-hour period PMP
        third24 = arcpy.da.SearchCursor(basinPMP, ("PMP_72",)).next()[0] -
arcpy.da.SearchCursor(basinPMP, ("PMP_48",)).next()[0]   # Calculate 3rd-largest 24-hour
period PMP

        arcpy.AddMessage("\n\tLargest 24-hour: " + str(largest24))
        arcpy.AddMessage("\tSecond largest 24-hour: " + str(second24))
        arcpy.AddMessage("\tThird largest 24-hour: " + str(third24))

        for distribution in distributionList:                              # Loops thourgh each 72-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            arcpy.AddMessage("\t\tFirst 24-hour Period...")
            accumPMP = 0
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to evenly distribute 2nd largest 24-hour
                for row in cursor:
```

```
                    if row[1] <= 96:                                # Leave loop once a row containing a
temporal dist. factor (ie, second 24h period) is reached
                        accumPMP +=  second24 / 96
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tSecond 24-hour Period...")
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to apply temporal factors to largest 24-hour PMP
                for row in cursor:
                    if row[1] > 96 and row[1] <= 192:                    # Constrain update to rows
97-192 (second 24hr period)
                        accumPMP = (largest24 * row[0]) + second24
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tThird 24-hour Period...")
            whereClause = distribution + " IS NULL"
            with arcpy.da.UpdateCursor(outTable, distribution, whereClause) as cursor:          #
Cursor to evenly distribute 3nd largest hour over remaining empty rows
                for row in cursor:
                    accumPMP +=  third24 / 96
                    row[0] = accumPMP
                    cursor.updateRow(row)
                del row, cursor, accumPMP, whereClause
            checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)

        if stormType == "Tropical":
            arcpy.AddMessage("\n***" + stormType + " Storm - 72hr PMP Temporal
Distributions***")
            temporalDistTable_72hr = home +
"\\Input\\Non_Storm_Data.gdb\\TS_TEMPORAL_DISTRIBUTIONS_72HR"    # Tropical
Storm Temporal distribution factors table
            outTable = outPath + "\\TS_Temporal_Distributions_72hr"
            arcpy.AddMessage("\n\tCreating temporal distribution table:...")
            dm.Copy(temporalDistTable_72hr, outTable)                    # Copy temporal dist.
factors table to output location
            distributionList = [field.name for field in arcpy.ListFields(temporalDistTable_72hr,
"TS*")]       # Create a list of 72-hour distribution field names
            arcpy.AddMessage("\n\tDistribution Field Names: " + str(distributionList))

            largest24 = arcpy.da.SearchCursor(basinPMP, ("PMP_24",)).next()[0]
# Calculate largest 24-hour period PMP
```

```
        second24 = arcpy.da.SearchCursor(basinPMP, ("PMP_48",)).next()[0] - largest24
# Calculate 2nd-largest 24-hour period PMP
        third24 = arcpy.da.SearchCursor(basinPMP, ("PMP_72",)).next()[0] -
arcpy.da.SearchCursor(basinPMP, ("PMP_48",)).next()[0]   # Calculate 3rd-largest 24-hour
period PMP

        arcpy.AddMessage("\n\tLargest 24-hour: " + str(largest24))
        arcpy.AddMessage("\tSecond largest 24-hour: " + str(second24))
        arcpy.AddMessage("\tThird largest 24-hour: " + str(third24))

        for distribution in distributionList:                    # Loops thourgh each 24-hour
temporal distribution
            arcpy.AddMessage("\n\tApplying temporal distribution for: " + distribution)
            arcpy.AddMessage("\t\tFirst 24-hour Period...")
            accumPMP = 0
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to evenly distribute 2nd largest hour
                for row in cursor:
                    if row[1] <= 96:                             # Leave loop once a row containing a
temporal dist. factor (ie, second 24h period) is reached
                        accumPMP +=  second24 / 96
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tSecond 24-hour Period...")
            with arcpy.da.UpdateCursor(outTable, [distribution, "TIMESTEP"]) as cursor:
# Cursor to apply temporal factors to largest 24-hour PMP
                for row in cursor:
                    if row[1] > 96 and row[1] <= 192:                # Constrain update to rows
97-192 (second 24hr period)
                        accumPMP = (largest24 * row[0]) + second24
                        row[0] = accumPMP
                        cursor.updateRow(row)
                del row, cursor

            arcpy.AddMessage("\t\tThird 24-hour Period...")
            whereClause = distribution + " IS NULL"
            with arcpy.da.UpdateCursor(outTable, distribution, whereClause) as cursor:          #
Cursor to evenly distribute 3nd largest hour over remaining empty rows
                for row in cursor:
                    accumPMP +=  third24 / 96
                    row[0] = accumPMP
                    cursor.updateRow(row)
                del row, cursor, accumPMP, whereClause
```

```
        checkTemporal(stormType, outPath, outTable, distributionList, dur, areaSize)
      return


   ##  This portion of the code checks to make sure none of the temporal distributions
   ##  are exceeding the PMP values for any durations.  It adds a table to the output
   ##  folder called CheckTemporal.
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


   def checkTemporal(stormType, outPath, TemporalTable, distributionFields, dur, areaSize):
      basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + areaSize
# Location of basin average PMP table
      pmpFields = [field.name for field in arcpy.ListFields(basinPMP, "PMP_*")]
# PMP duration run
      temporalFields = [field.name for field in arcpy.ListFields(TemporalTable)]
      table = arcpy.Describe(TemporalTable)
      tableName = table.name


      pmp = []                                                  #Creates empty list and
updates with PMP values for each duration run
      i = 0
      while i < len(pmpFields):
         with arcpy.da.SearchCursor(basinPMP,pmpFields) as cursor:
            for row in cursor:
               pmp.append(row[i])
               i += 1
      del i, cursor

      checkTable = outPath + "\\Temporal_Distribution_Check_" + stormType
      arcpy.AddMessage("\nCheckTable: "  + checkTable)
      maxFields = []                                              #Create Max fields
for each duration
      checkFields = []                                            #Create Check fields
for each duration
      if arcpy.Exists(checkTable):
         with arcpy.da.InsertCursor(checkTable, "PATTERN") as cursor:
            for val in distributionFields:
               cursor.insertRow([val])
         i = 0                                                    #Populate fields
         for pmpField in pmpFields:
            with arcpy.da.UpdateCursor(checkTable, pmpField) as cursor:
               for row in cursor:
                  row = pmp[i]
```

```
                cursor.updateRow([row])
            i += 1
        del i, cursor
    else:
        checkTable = dm.CreateTable(outPath, "Temporal_Distribution_Check_" + stormType)
#Creates table in output folder, adds field, and populates field with distributions
        dm.AddField(checkTable, "PATTERN", "TEXT", "", "", 50)
        with arcpy.da.InsertCursor(checkTable, "PATTERN") as cursor:
            for val in distributionFields:
                cursor.insertRow([val])
        for maxField in pmpFields:
            newField = maxField.replace("PMP","MAX")
            maxFields.append(newField)
        del newField
        for checkField in pmpFields:
            newField = checkField.replace("PMP","CHECK")
            checkFields.append(newField)
        del newField
        i = 0                                                    #Populate fields
        for pmpField in pmpFields:
            dm.AddField(checkTable, pmpField, "DOUBLE", "", "", 50)
            dm.AddField(checkTable, maxFields[i], "DOUBLE", "", "", 50)
            dm.AddField(checkTable, checkFields[i], "TEXT", "", "", 50)
            with arcpy.da.UpdateCursor(checkTable, pmpField) as cursor:
                for row in cursor:
                    row = pmp[i]
                    cursor.updateRow([row])
                i += 1
        del i, cursor

    step = arcpy.da.SearchCursor(TemporalTable,("MINUTE",)).next()[0]
    if step == 15:
        dic = {"01": 4, "02": 8, "03": 12, "04": 16, "05": 20, "06": 24, "12": 48, "24": 96, "48":
192, "72": 288, "96": 384, "120": 480}  # Dictionary to convert durations into 15-minute
timesteps
    elif step == 5:
        dic = {"01": 12, "02": 24, "03": 36, "04": 48, "05": 60, "06": 72, "12": 144, "24": 288,
"48": 576, "72": 864, "96": 1152, "120": 1440}
    elif step == 60:
        dic = {"01": 1, "02": 2, "03": 3, "04": 4, "05": 5, "06": 6, "12": 12, "24": 24, "48": 48,
"72": 72, "96": 96, "120": 120}
    # arcpy.AddMessage(str(step) + " Minute distribution Pattern.....")

    maxFields = [field.name for field in arcpy.ListFields(checkTable, "MAX*")]
    i = 0                                # Calculates incremental PMP depths from temporal
distribution and gets maximum rainfall for each duration run
```

```
        d = durList.index(dur) + 1
        for dur in durList[:d]:
            k = dic[dur]
            p = 3                                                    # Skip first 3 fields in
temporaltable (objectID, Timesteps, minutes)
            for distribution in distributionFields:
                incPMP = []
                previousRow = 0
                with arcpy.da.SearchCursor(TemporalTable, temporalFields) as cursor:
                    for row in cursor:
                        increment = row[p] - previousRow
                        previousRow = row[p]
                        incPMP.append(increment)
                na = np.array(incPMP)
                sumList = np.convolve(na,np.ones(k))
                maxPMP = max(sumList)
                maximumPMP = math.trunc(maxPMP * 10 ** 2.0) / 10 ** 2.0
                p += 1

                x = 0
                with arcpy.da.UpdateCursor(checkTable, ["PATTERN", maxFields[i]]) as cursor:
# Updates table with max values
                    for row in cursor:
                        if row[0] == distribution:
                            row[1] = maximumPMP
                            x += 1
                            cursor.updateRow(row)
            i += 1
        del i, k, cursor, x
        with arcpy.da.UpdateCursor(checkTable, '*') as cursor:                # Compares PMP
values to max values for each duration.  If PMP values are larger update check field with PASS
if not FAIL
            for row in cursor:
                rec = dict(zip(cursor.fields, row))
                arcpy.AddMessage("\n\n\tChecking temporally distributed depth-durations against
PMP: " + rec['PATTERN'] + "\n")
                for k, v in rec.items():
                    if not k.startswith('PMP_'):
                        continue
                    _, n = k.split('_')
                    try:
                                        # This try/except skips comparisons for additional durations not
present in current temporal pattern
                        mx = rec['MAX_{}'.format(n)]
                        rec['CHECK_{}'.format(n)] = 'FAIL' if v < mx else 'PASS'
                    except:
```

```
                arcpy.AddMessage("\n\tDuration not present...")
                continue
            if rec['CHECK_{}'.format(n)] == 'PASS':
                arcpy.AddMessage("\t" + str(n) + "-hour \n\t\tPMP value is... " + str(v) + "
\n\t\tmax rainfall value is..." + str(mx) + "\n\t\tThis distribution.... " +
rec['CHECK_{}'.format(n)])
            else:
                arcpy.AddMessage("\t" + str(n) + "-hour \n\t\tPMP value is... " + str(v) + "
\n\t\tmax rainfall value is..." + str(mx) + "\n\t\tThis distribution.... " +
rec['CHECK_{}'.format(n)]+ "\n\t\t**Max values for duration are exceeding PMP values. Use of
this temporal distribtion not recommended.")
            cursor.updateRow([rec[k] for k in cursor.fields])
    del cursor, k, v, rec
    return


  ################################################################################
  ##  The temporalCritStacked() function applies the critically stacked
  ##  temporal distributions scenarios.  The function accepts the storm type,
  ##  output .gdb path, AOI area size, PMP duration string (hours), and
  ##  integer timestep duration (minutes). The function outputs a gdb table.


  def temporalCritStacked(stormType, outPath, area, duration, timestep):
# Function applied Critically Stacked temporal distribution
    basinPMP = outPath + "\\" + stormType + "_PMP_Basin_Average_" + area
# Location of basin average PMP table
    if stormType == "Local" and duration == "06":
# These conditional statements define the field name based on storm type, PMP duration, and
timestep duration
        csField = "LS_" + duration + "_HOUR_" + str(timestep) + "MIN_CRIT_STACKED"
    elif stormType == "General":
        csField = "GS_" + duration + "_HOUR_" + str(timestep) + "MIN_CRIT_STACKED"
    elif stormType == "Tropical":
        csField = "TS_" + duration + "_HOUR_" + str(timestep) + "MIN_CRIT_STACKED"
    else:
        arcpy.AddMessage("\n***Invalid storm type: " + stormType)
        return
    arcpy.AddMessage("\n***" + duration + "-hour " + str(timestep) + "-min Critically Stacked
Temporal Distribution***")
    tableName = "Temporal_Distribution_" + duration + "hr_" + str(timestep) +
"min_Crit_Stacked"                          # Output table name
    tablePath = outPath + "\\" + tableName
# Output table full path
    pmpFields = [field.name for field in arcpy.ListFields(basinPMP, "PMP*")]
# Gets the "PMP_XX" field names from the basin avereage PMP table
```

```
        if duration == "06":                                                    #
These conditional statements define the key durations needed to build the critically stacked
patterns for the following durations...
            keyDurations = [1, 2, 3, 4, 5, 6]
        elif duration == "12":
            keyDurations = [1, 2, 3, 4, 5, 6, 12]
        elif duration == "24":
            keyDurations = [1, 2, 3, 4, 5, 6, 12, 24]
        elif duration == "48":
            keyDurations = [1, 2, 3, 4, 5, 6, 12, 24, 48]
        elif duration == "72":
            keyDurations = [1, 2, 3, 4, 5, 6, 12, 24, 48, 72]
        elif duration == "96":
            keyDurations = [1, 2, 3, 4, 5, 6, 12, 24, 48, 72, 96]
        elif duration == "120":
            keyDurations = [1, 2, 3, 4, 5, 6, 12, 24, 48, 72, 96, 120]
        else:
            arcpy.AddMessage("\n\t...Critically stacked temporal distribution not available for " +
duration + "-hour duration.")
            return
        timestepLen = int(duration) * 60 // timestep
# number of rows in output table
        xValues = [0]
        for i in keyDurations:                                                  #
defines the known x-values (xp) to be used in the interpolation
            xVal = i * timestepLen // int(duration)
            xValues.append(xVal)
        del i, xVal
        yValues = [0]
        d = 0
        for i in keyDurations:                                                  #
defines the known y-values (fp) to be used in the interpolation
            pmpDepth = arcpy.da.SearchCursor(basinPMP, pmpFields).next()[d]
            yValues.append(pmpDepth)
            d += 1
        del d, i, pmpDepth

        x = np.arange(0, timestepLen + 1, 1)
# defines the x points at which to interpolate values
        xp = np.asarray(xValues)                                                #
np.asarray converts lists into numpy arrays
        fp = np.asarray(yValues)
        y = np.interp(x, xp, fp)
        inc = []
        prevDepth = 0
```

```
        i = 0
        for depth in np.nditer(y):                                    #
populates incremental depths list 'inc' with y array
            inc.append(depth - prevDepth)
            prevDepth = depth
            i += 1
        del i, prevDepth
        periods = int(duration)                                       #
defines number of periods (known hours) as the duration
        periodLen = 60 // timestep                                      #
defines number of timesteps (minutes) in each period
        ranks = []
        stackRank = 1
        i = 0
        while i < periods:                                            #
populates list 'ranks' with a rank integer, one entry per period
            ranks.append(stackRank)
            stackRank += periodLen
            i += 1
        del i

        orderRanks = []
        orderRanks.insert(0, ranks.pop(0))
        for i in range (timestepLen // periodLen):
## orders the ranks according to critically stacked pattern.  Pulls
            if ranks:                                                 ## (pop())
the first rank from the ranks list and places it in the orderRanks
                orderRanks.insert(0, ranks.pop(0))
## list. Places next two ranks at the beginning of the list
            if ranks:                                                 ## and the
following at the end of the list.  Repeats until ranks is empty.
                orderRanks.insert(0, ranks.pop(0))
            if ranks:
                orderRanks.append(ranks.pop(0))
        del i
        orderRanks += [orderRanks.pop(0)]
        if orderRanks[0] == max(orderRanks):
## Moves last rank to the end of of orderRanks list.
            arcpy.AddMessage("\n*** moving first rank to last")
            orderRanks.append(orderRanks.pop(max))
        orderInc = []
        n = 0
        for i in range(periods):                                      # gets
the nth largest increment where n is the ordered Rank.
            for q in range(periodLen):
                nthLargest = nlargest(orderRanks[n], inc)[-1]
```

```
            orderInc.append(nthLargest)
        n += 1
    del n, i, q
    cumulative = []
    prevInc = 0
    for i in orderInc:                                                    #
Converts the incremental depths to cumulatove depths and places in cumulative list
        value = round(i + prevInc, 2)
        cumulative.append(value)
        prevInc = i + prevInc
        i += 1
    del i, prevInc
    timesteps = x.tolist()                                                #
Converts the timesteps array (x) to a list then removes the first zero entry
    timesteps.pop(0)
    minutes = []
    minutesInc = timestep
    for i in range(timestepLen):                                          #
Constructs the minutes list to be used in output column based on timestep interval
        minutes.append(minutesInc)
        minutesInc += timestep
    del i
    dm.CreateTable(outPath, tableName)
# Create the output geodatabase table
    dm.AddField(tablePath, "TIMESTEP", "DOUBLE")
# Create "TIMESTEP" field
    dm.AddField(tablePath, "MINUTES", "DOUBLE")
# Create "MINUTES" field
    dm.AddField(tablePath, csField, "DOUBLE")
# Create cumulated rainfall field
    zipped = zip(timesteps, minutes, cumulative)
# Zip up lists of output items.
    fields = ('TIMESTEP', 'MINUTES', csField)
# Output table field names
    arcpy.AddMessage("\n\tApplying temporal distribution for: " + csField)
    with arcpy.da.InsertCursor(tablePath, fields) as cursor:
# Cursor to populate output Critically Stacked table
        for i in zipped:
            cursor.insertRow(i)
    del cursor, i
    return


##################################################################################
##  This portion of the code iterates through each storm feature class in the
##  'Storm_Adj_Factors' geodatabase (evaluating the feature class only within
```

```
    ##  the Local, Tropical, or general feature dataset).  For each duration,
    ##  at each grid point within the aoi basin, the transpositionality is
    ##  confirmed.  Then the DAD precip depth is retrieved and applied to the
    ##  total adjustement factor to yield the total adjusted rainfall.  This
    ##  value is then sent to the updatePMP() function to update the 'PMP_Points'
    ##  feature class.
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~##


    desc = arcpy.Describe(basin)                                    # Check to ensure AOI input
shape is a Polygon. If not - exit.
    basinShape = desc.shapeType
    if desc.shapeType == "Polygon":
        arcpy.AddMessage("\nBasin shape type: " + desc.shapeType)
    else:
        arcpy.AddMessage("\nBasin shape type: " + desc.shapeType)
        arcpy.AddMessage("\nError: Input shapefile must be a polygon!\n")
        sys.exit()

    createPMPfc()                                              # Call the createPMPfc() function
to create the PMP_Points feature class.

    env.workspace = adjFactGDB                                        # the workspace
environment is set to the 'Storm_Adj_Factors' file geodatabase

    aoiSQMI = round(getAOIarea(),2)                                  # Calls the getAOIarea()
function to assign area of AOI shapefile to 'aoiSQMI'
    if aoiSQMI > 100 and stormType is "Local":
        arcpy.AddMessage("\n***Warning - Local storm PMP depths only valid for basins 100
square miles or smaller***")

    stormList = arcpy.ListFeatureClasses("", "Point", stormType)                # List all the total
adjustment factor feature classes within the storm type feature dataset.
    for dur in durList:

arcpy.AddMessage("\n*********************************************************
**\nEvaluating " + dur + "-hour duration...")

        pmpList = []
        driverList = []
        gridRows = arcpy.SearchCursor(env.scratchGDB + "\\PMP_Points")
        try:
            for row in gridRows:
```

```
            pmpList.append(0.0)                                    # creates pmpList of empty
float values for each grid point to store final PMP values
            driverList.append("STORM")                             # creates driverList of
empty text values for each grid point to store final Driver Storm IDs
         del row, gridRows
      except UnboundLocalError:
         arcpy.AddMessage("\n***Error: No data present within basin/AOI area.***\n")
         sys.exit()

      env.workspace = adjFactGDB
      for storm in stormList[:]:
         arcpy.AddMessage("\n\tEvaluating storm: " + storm + "...")
         dm.MakeFeatureLayer(storm, "stormLayer")                  # creates a feature
layer for the current storm
         dm.SelectLayerByLocation("stormLayer", "HAVE_THEIR_CENTER_IN", "vgLayer")
# examines only the grid points that lie within the AOI
         gridRows = arcpy.SearchCursor("stormLayer")
         pmpField = "PMP_" + dur
         i = 0
         try:
            dadPrecip = round(dadLookup(storm, dur, aoiSQMI),3)
            arcpy.AddMessage("\t\t" + dur + "-hour DAD value:  " + str(dadPrecip) + chr(34))
         except TypeError:                                         # In no duration exists in the DAD
table - move to the next storm
            arcpy.AddMessage("\t***Duration '" + str(dur) + "-hour' is not present for " +
str(storm) + ".***\n")
            continue
         arcpy.AddMessage("\t\tComparing " + storm + " adjusted rainfall values against current
driver values...")
         transCounter = 0                                          # Counter for number of grid points
transposed to
         for row in gridRows:
            if row.TRANS == 1:                                     # Only continue if grid point is
transpositionable ('1' is transpostionable, '0' is not).
               try:                                                # get total adj. factor if duration exists
                  transCounter += 1
                  adjRain = round(dadPrecip * row.TAF,1)
                  if adjRain > pmpList[i]:
                     pmpList[i] = adjRain
                     driverList[i] = storm
               except RuntimeError:
                  arcpy.AddMessage("\t\t   *Warning*  Total Adjusted Raifnall value falied to set
for row " + str(row.CNT))
                  break
               del adjRain
            i += 1
```

```
            if transCounter == 0:
                arcpy.AddMessage("\t\tStorm not transposable to basin. Removing " + storm + " from
list...\n")
                stormList.remove(storm)
            else:
                arcpy.AddMessage("\t\tTransposed to " + str(transCounter) + "/" + str(i) + " grid
points...\n")
            del row, transCounter
        del storm, gridRows, dadPrecip
        updatePMP(pmpList, driverList, dur)             # calls function to update "PMP Points"
feature class
    del pmpList, stormList

    arcpy.AddMessage("\n'PMP_Points' Feature Class 'PMP_XX' fields update complete for all '"
+ stormType + "' storms.")

    outputPMP(stormType, aoiSQMI, outputPath)             # calls outputPMP() function
    outArea = str(int(round(aoiSQMI,0))) + "sqmi"
    outGDB = outLocation + "\\" + stormType + "\\PMP_" + desc.baseName + "_" + outArea +
".gdb"
    basinName = desc.baseName

    if runTemporal:                                 #Calls temporal distribution functions
        centroidLocation = basinZone(basin)
        arcpy.AddMessage("\nBasin Centroid Transposition Zone: " + str(centroidLocation))

        for dur in durList:
            if dur == "02":
                temporalDistLS2(stormType, outGDB, centroidLocation, outArea)
            if dur == "06":
                temporalDistLS(stormType, outGDB, centroidLocation, outArea)
                temporalDistControlStorm_06hr(stormType, outGDB, centroidLocation, outArea,
basinName)
            if stormType == "Local" and dur == "06":
                temporalCritStacked(stormType, outGDB, outArea, dur, 5)
            if dur == "12":
                temporalDistControlStorm_12hr(stormType, outGDB, centroidLocation, outArea,
basinName)
            if stormType == "Local" and dur == "12":
                temporalDist_LS12hr(stormType, outGDB, centroidLocation, outArea)
            if dur == "24":
                temporalDist_24hr(stormType, outGDB, centroidLocation, outArea)
                temporalCritStacked(stormType, outGDB, outArea, dur, 15)
                temporalDistControlStorm_24hr(stormType, outGDB, centroidLocation, outArea,
basinName)
            if dur == "48":
```

```
            temporalDist_48hr(stormType, outGDB, centroidLocation, outArea)
        if dur == "72":
            temporalDist_72hr(stormType, outGDB, centroidLocation, outArea)
            temporalDistControlStorm_72hr(stormType, outGDB, centroidLocation, outArea,
basinName)


    i = 0                                                        #Creates CSV files of all output tables
    csvPath = outLocation + "\\" + stormType + "\\CSV_" + desc.baseName + "_" + outArea +
"\\"
    if not arcpy.Exists(outLocation + "\\" + stormType + "\\CSV_" + desc.baseName + "_" +
outArea):
        arcpy.CreateFolder_management(outLocation + "\\" + stormType + "\\", "CSV_" +
desc.baseName + "_" + outArea)
    arcpy.AddMessage("\n\t...Creating output tables as CSV files.. ")
    env.workspace = outGDB
    outTables = arcpy.ListTables()
    for t in outTables:
        arcpy.TableToTable_conversion(t, csvPath, outTables[i] + ".csv")
        i += 1
    xmlFiles = os.listdir(csvPath)
    for file in xmlFiles:
        if file.endswith(".xml"):
            os.remove(os.path.join(csvPath,file))
    return
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~##


def outputBasAveTable():
    arcpy.AddMessage("\nCreating basin average summary table.\n")
    tableList = basAveTables
    for table in tableList:
        arcpy.AddMessage("\t\tMerging tables... " + table)

    dm.Merge(basAveTables, outputTable)
    ##  addLayerMXD(outputTable)  adds output table to ArcMap session

    return
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~##


def addLayerMXD(addFC):
```

```
    desc = arcpy.Describe(addFC)
    layerName = desc.name
    arcpy.AddMessage("\nAdding " + layerName + " table to current MXD...")
    if desc.dataType == "FeatureClass":
        dm.MakeFeatureLayer(addFC, layerName)
        layer = arcpy.mapping.Layer(layerName)
        arcpy.mapping.AddLayer(df, layer)
        arcpy.AddMessage("\n" + layerName + " added to current map session.\n")
    elif desc.dataType == "Table":
        layer = arcpy.mapping.TableView(desc.catalogPath)
        arcpy.mapping.AddTableView(df, layer)
        arcpy.AddMessage("\n" + layerName + " added to current map session.\n")
    elif desc.dataType == "ArcInfoTable":
        layer = arcpy.mapping.TableView(desc.catalogPath + ".dbf")
        arcpy.mapping.AddTableView(df, layer)
        arcpy.AddMessage("\n" + layerName + " added to current map session.\n")

    del desc, layerName, layer
    return


##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~##


if locDurations:
    type = "Local"
    durations = locDurations
    dm.CreateFolder(outLocation, type)
    outputPath = outLocation + "\\Local\\"
    arcpy.AddMessage("\nRunning PMP analysis for storm type: " + type)
    pmpAnalysis(basin, type, durations)          # Calls the pmpAnalysis() function to calculate the
local storm PMP
    arcpy.AddMessage("\nLocal storm analysis
complete...\n***************************************************************
************************************")

if genDurations:
    type = "General"
    durations = genDurations
    dm.CreateFolder(outLocation, type)
    outputPath = outLocation + "\\General\\"
    arcpy.AddMessage("\nRunning PMP analysis for storm type: " + type)
    pmpAnalysis(basin, type, durations)          # Calls the pmpAnalysis() function to calculate the
general storm PMP
```

```
    arcpy.AddMessage("\nGeneral storm analysis
complete...\n**********************************************************************
**************************************")

if tropDurations:
    type = "Tropical"
    durations = tropDurations
    dm.CreateFolder(outLocation, type)
    outputPath = outLocation + "\\Tropical\\"
    arcpy.AddMessage("\nRunning PMP analysis for storm type: " + type)
    pmpAnalysis(basin, type, durations)          # Calls the pmpAnalysis() function to calculate the
tropical storm PMP
    arcpy.AddMessage("\nTropical storm analysis
complete...\n**********************************************************************
**************************************")

#if arcpy.Describe(outputTable).name:
#    outputBasAveTable()

#arcpy.RefreshTOC()
#arcpy.RefreshActiveView()
#del mxd, df
```