ArcFootprint

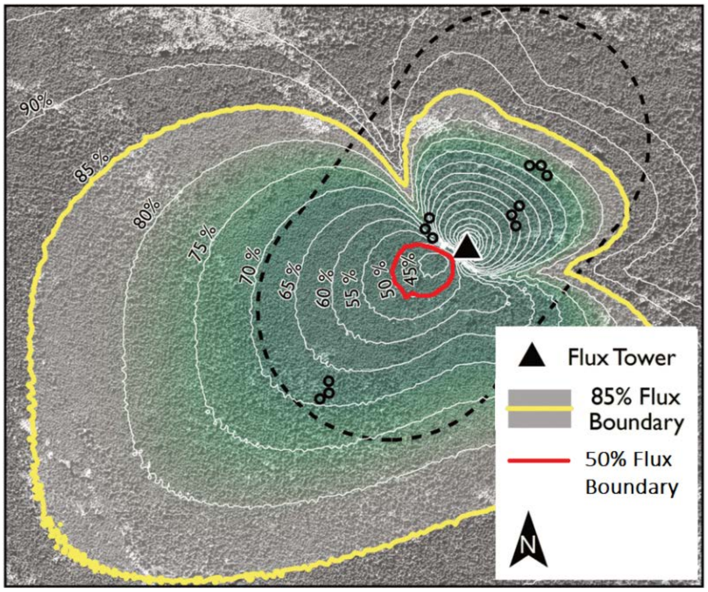
A flux footprint estimation tool for ArcGIS Desktop

Mads O'Brien
FES754: Geospatial Software Design
December 2019

Table of Contents

Introduction

What is a flux footprint?



When earth scientists attempt to estimate greenhouse gas emissions, they aim to calculate the **flux** of a substance—the amount of that substance flowing from a unit area over a unit time. A question that may arise is, "*Where* did this carbon dioxide I'm estimating come *from*, exactly?"

One solution: Generate a **flux footprint**, a polygon describing the source area of gas, heat, or water given off by a surface that is detected by a sensor. For example, in the figure at right,[1] the red line designates that 50% of the instantaneous gas measurements taken by a sensor atop a tower have originated from this region on the ground and have been blown to the sensor by the wind. Similarly, 85% of what the sensor "sees" lies within the yellow boundary. Variables like wind speed and wind direction can be used to "back-calculate" where gusts of air measured by the sensor have originated from.

Numerous software programs exist for estimating flux footprints from observational gas data.[2] However, many of these footprint estimation tools:

1) **rely on proprietary or standalone software**, often sold by the same companies that manufacture the (pricey) instruments used to collect flux measurements;
2) **require relatively advanced programming skills**, particularly when the tools are free or open-source;
3) **do not generate outputs compatible with common GIS software** like ArcMap, and multi-step conversions increase the lag time between footprint calculation and comparing footprint extent with other spatial datasets.

Goals of this project

- Build a user-friendly, point-and-click ArcToolbox tool for generating spatial extents of flux footprints (as shapefiles), based upon meteorological variables collected from gas sensors.
- Construct separate polygons for each source area (50%, 90%, etc.), making it easy for the user to run subsequent Zonal Statistics within each zone.
- Create a tool that will not only benefit my own academic research, but hopefully benefit others conducting meteorological research as well.

---

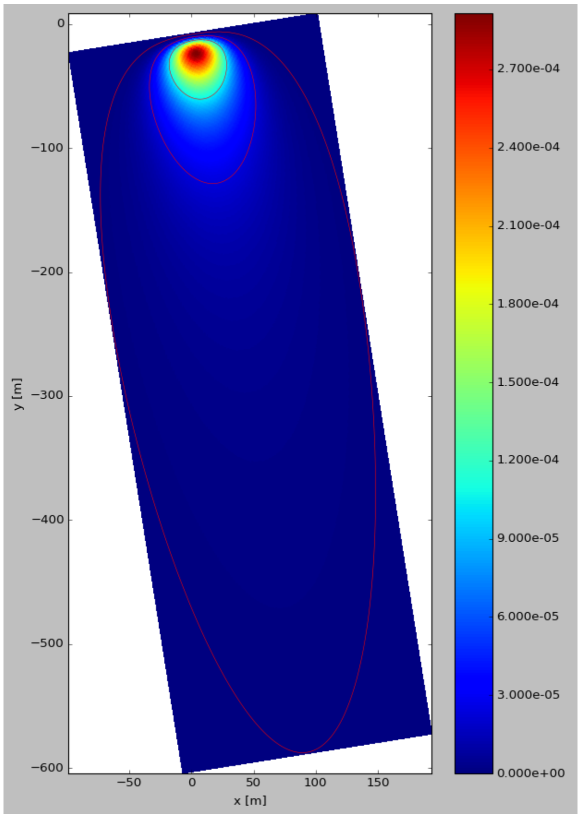
[1] Figure adapted from Ferster, C.J., Trofymow, J.A., Coops, N.C., Chen, B., Black, T.A. and Gougeon, F.A., 2011. Determination of ecosystem carbon-stock distributions in the flux footprint of an eddy-covariance tower in a coastal forest in British Columbia. Canadian journal of forest research, 41(7), pp.1380-1393.

[2] See list of examples at fluxnet.fluxdata.org/2017/10/10/toolbox-a-rolling-list-of-softwarepackages-for-flux-related-data-processing/

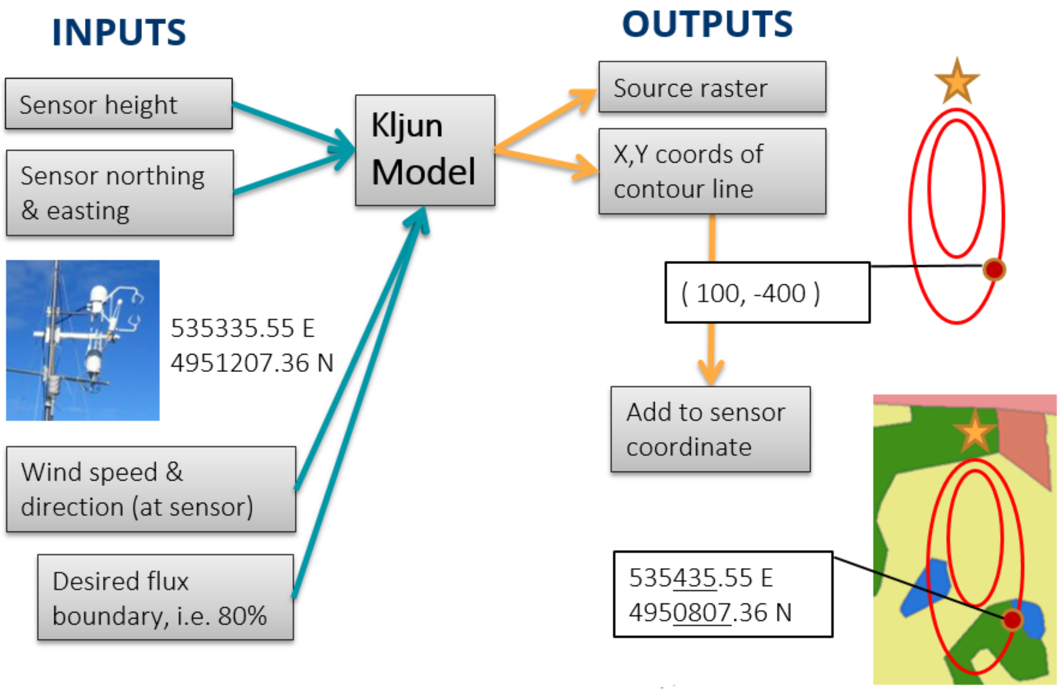
Methods

Scholars have formulated a variety of mathematical methods to estimate flux footprints. ArcFootprint relies upon the widely-cited method[3] developed by **Kljun et al (2015)**,[4] which has (conveniently) already been translated into a Python script.[5] Essentially, the script defines a single function, called ***FFP( )*** for **Flux Footprint Prediction**, that a user may insert in their own program.

The Kljun method's calculations generate a (non-georeferenced) raster, where each pixel value represents a percentage contribution—i.e., a single dark red pixel contributes 0.000072% of the gas "seen" by a sensor. This percent contribution raster is then the basis for drawing "cumulative contour lines": starting from the highest pixel value, one can imagine an ellipse radiating outward until all the pixels within that ellipse add up to [XX]%



Simplified flowchart of ArcFootprint method



---

[3] The Kljun model is the same one used in Tovi, a major Proprietary software for footprint modeling.

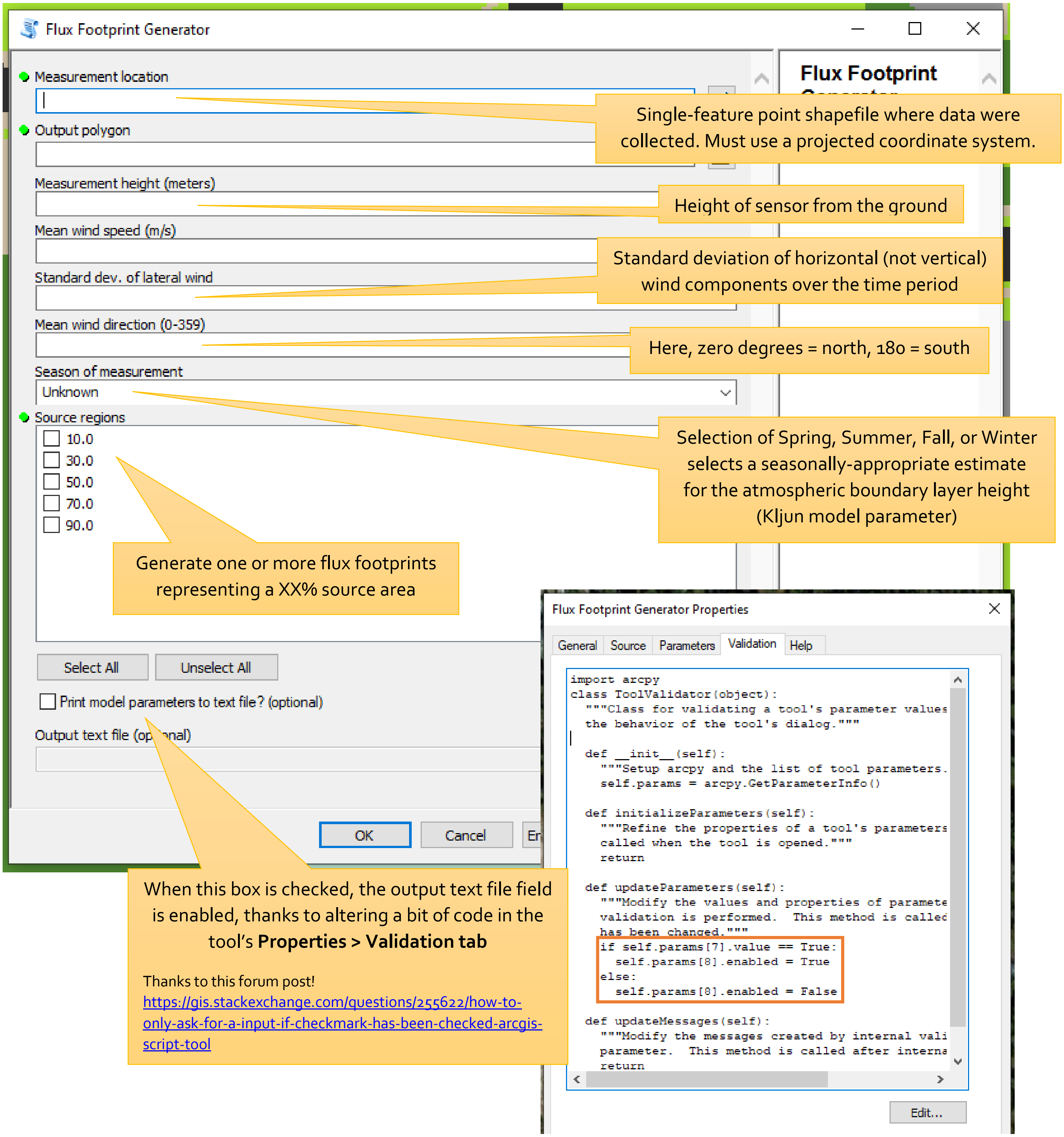
[4] Kljun, N., P. Calanca, M.W. Rotach, H.P. Schmid, 2015: A simple two-dimensional parameterisation for Flux Footprint Prediction (FFP). Geosci. Model Dev., 8, 3695-3713. doi:10.5194/gmd-8-3695-2015.

[5] To download the original Flux Footprint Prediction source code, visit http://footprint.kljun.net/download_2.php

Tool inputs

First, a user must acquire gas observation data. One can download freely-available flux tower datasets from websites like https://fluxnet.fluxdata.org/ or https://ameriflux.lbl.gov/. Alternatively, a user could experiment with other above-the-ground gas sensors, such as measurements acquired from a UAV. Ideally, a single footprint is estimated from data averaged over a half-hour period.

No matter how gas concentration data were collected, the user must possess the following variables:



Single-feature point shapefile where data were collected. Must use a projected coordinate system.

Height of sensor from the ground

Standard deviation of horizontal (not vertical) wind components over the time period

Here, zero degrees = north, 180 = south

Selection of Spring, Summer, Fall, or Winter selects a seasonally-appropriate estimate for the atmospheric boundary layer height (Kljun model parameter)

Generate one or more flux footprints representing a XX% source area

When this box is checked, the output text file field is enabled, thanks to altering a bit of code in the tool's **Properties > Validation tab**

Thanks to this forum post! https://gis.stackexchange.com/questions/255622/how-to-only-ask-for-a-input-if-checkmark-has-been-checked-arcgis-script-tool

```
import arcpy
class ToolValidator(object):
  """Class for validating a tool's parameter values
  the behavior of the tool's dialog."""

  def __init__(self):
    """Setup arcpy and the list of tool parameters.
    self.params = arcpy.GetParameterInfo()

  def initializeParameters(self):
    """Refine the properties of a tool's parameters
    called when the tool is opened."""
    return

  def updateParameters(self):
    """Modify the values and properties of paramete
    validation is performed.  This method is called
    has been changed."""
    if self.params[7].value == True:
      self.params[8].enabled = True
    else:
      self.params[8].enabled = False

  def updateMessages(self):
    """Modify the messages created by internal vali
    parameter.  This method is called after interna
    return
```

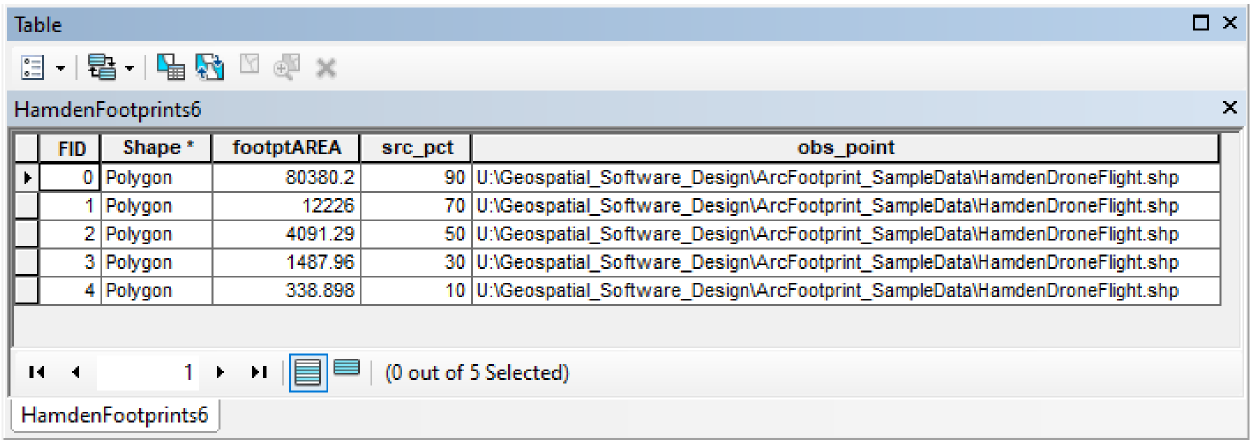
Tool outputs

Background outputs (not shown to user)

1) **Percent contribution raster** generated by FFP() function, like the one pictured on page 3
2) **xrs** and **yrs**, arrays of footprint boundary coordinates relative to a (0,0) origin. After translating these "relative" coordinates to "on the Earth around the sensor" coordinates, the script "connects the dots" to draw ellipses from these coordinates.

User outputs

1) **The footprint polygon shapefile**. When a user requests more than one source footprint to be generated, each ellipse is drawn one at a time. Thus, the polygons overlap with each other, and the area of each feature accurately represents the entire source area, not just the non-overlapping portion.
2) **Additional descriptive fields**. The following fields are added to each output attribute table:



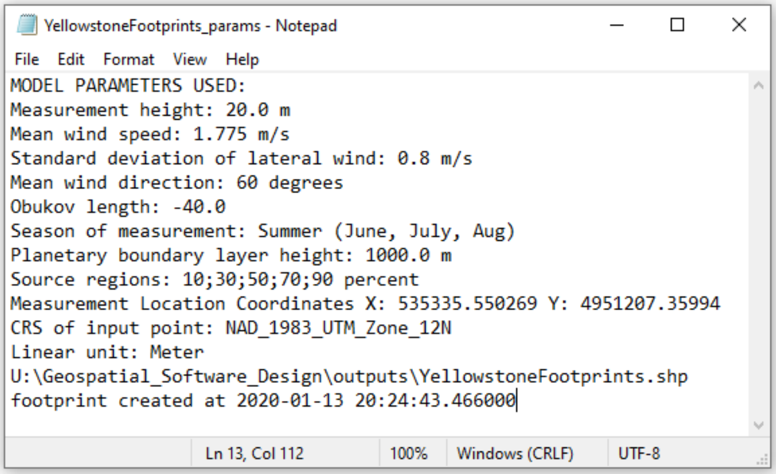
FID	Shape *	footptAREA	src_pct	obs_point
0	Polygon	80380.2	90	U:\Geospatial_Software_Design\ArcFootprint_SampleData\HamdenDroneFlight.shp
1	Polygon	12226	70	U:\Geospatial_Software_Design\ArcFootprint_SampleData\HamdenDroneFlight.shp
2	Polygon	4091.29	50	U:\Geospatial_Software_Design\ArcFootprint_SampleData\HamdenDroneFlight.shp
3	Polygon	1487.96	30	U:\Geospatial_Software_Design\ArcFootprint_SampleData\HamdenDroneFlight.shp
4	Polygon	338.898	10	U:\Geospatial_Software_Design\ArcFootprint_SampleData\HamdenDroneFlight.shp

footptAREA – the area covered by the source region, calculated in the linear units used by the footprint shapefile's CRS (meters, feet, etc.)

src_pct – source percent; identifies each footprint feature with the source area it signifies (10%, 30%, etc.)

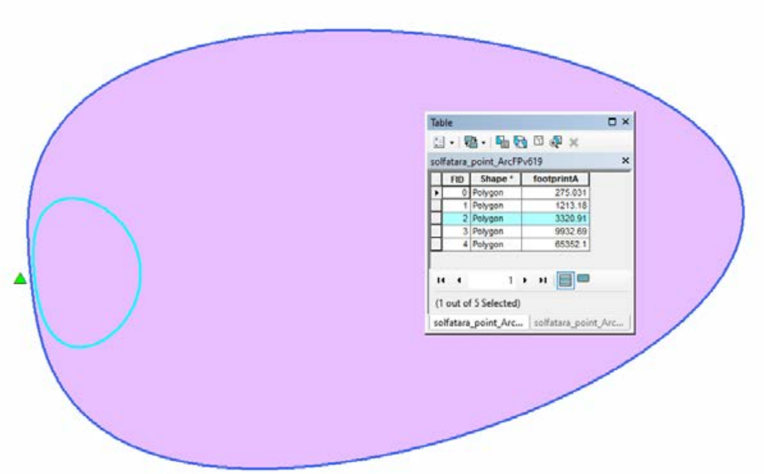
obs_point – observation point; records point shapefile used as input to generate the footprint

If requested**, a text file** recording model parameters used in the footprint estimation is written.



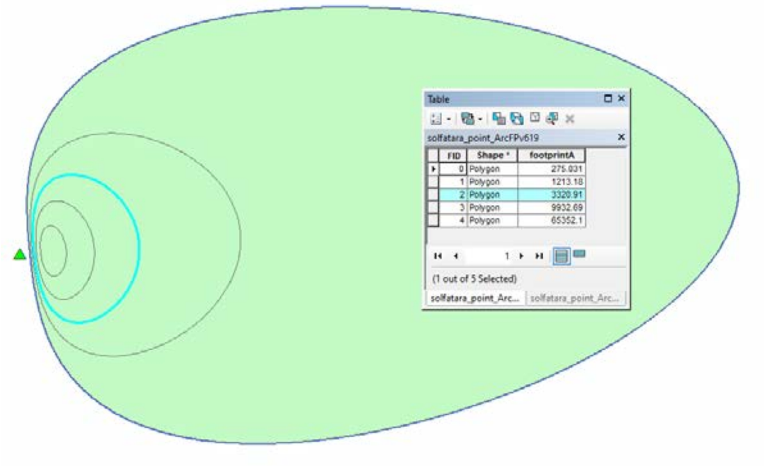
```
MODEL PARAMETERS USED:
Measurement height: 20.0 m
Mean wind speed: 1.775 m/s
Standard deviation of lateral wind: 0.8 m/s
Mean wind direction: 60 degrees
Obukov length: -40.0
Season of measurement: Summer (June, July, Aug)
Planetary boundary layer height: 1000.0 m
Source regions: 10;30;50;70;90 percent
Measurement Location Coordinates X: 535335.550269 Y: 4951207.35994
CRS of input point: NAD_1983_UTM_Zone_12N
Linear unit: Meter
U:\Geospatial_Software_Design\outputs\YellowstoneFootprints.shp
footprint created at 2020-01-13 20:24:43.466000
```

Other tool features



	FID	Shape *	footprintA
▸	0	Polygon	275.031
	1	Polygon	1213.18
	2	Polygon	3320.91
	3	Polygon	9932.69
	4	Polygon	65352.1

(1 out of 5 Selected)



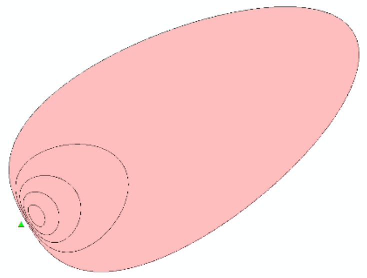
	FID	Shape *	footprintA
▸	0	Polygon	275.031
	1	Polygon	1213.18
	2	Polygon	3320.91
	3	Polygon	9932.69
	4	Polygon	65352.1

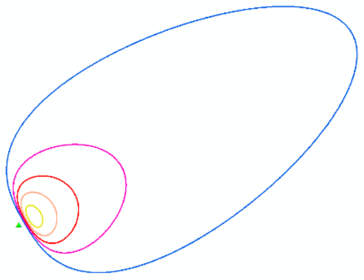
(1 out of 5 Selected)

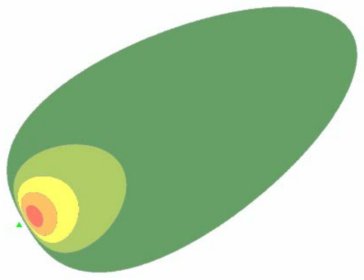
Polygon drawing order to maximize visibility

ArcMap renders shapes in the order that they are listed within an attribute table. This means that a larger polygon can overlap and entirely obscure a smaller polygon (see purple shapefile). ArcFootprint sorts the footprint polygons in descending order by size; thus, smaller polygons are drawn later and on top of larger ones (see green shapefile).

Each polygon is complete (has no holes), meaning shape areas are cumulative and symbolization is easy. (See below!)

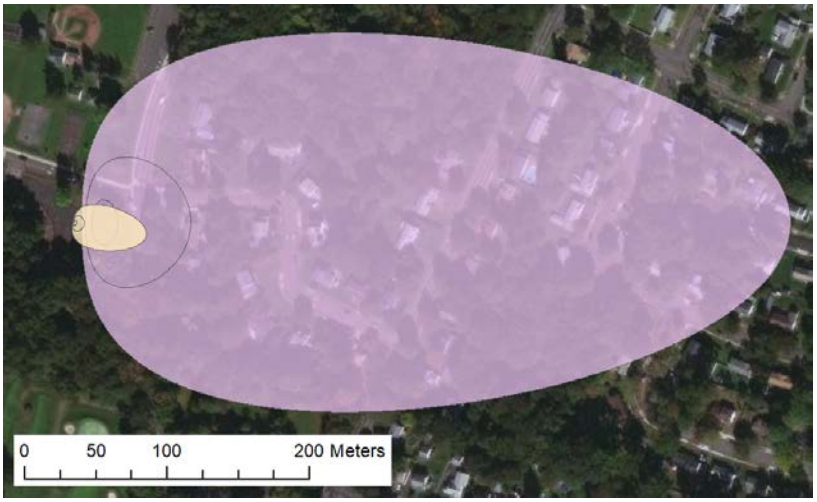






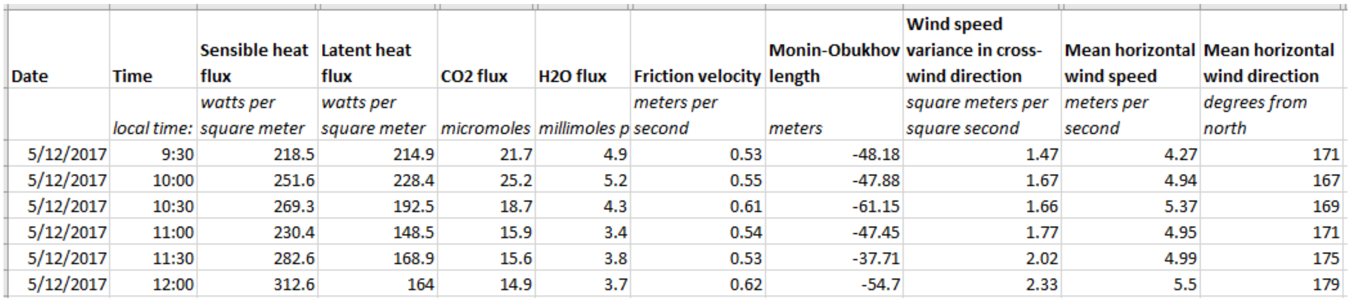
Auto-conversion of linear units

The Kljun model generates footprint boundaries using relative *meter* coordinates, at first. If a footprint vertex should be plotted 100m from a sensor, but the input sensor location shapefile uses US Feet as a linear unit, ArcFootprint will correct the relative coordinates and ensure the vertex is plotted not 100 feet from the sensor, but the proper 328 feet.

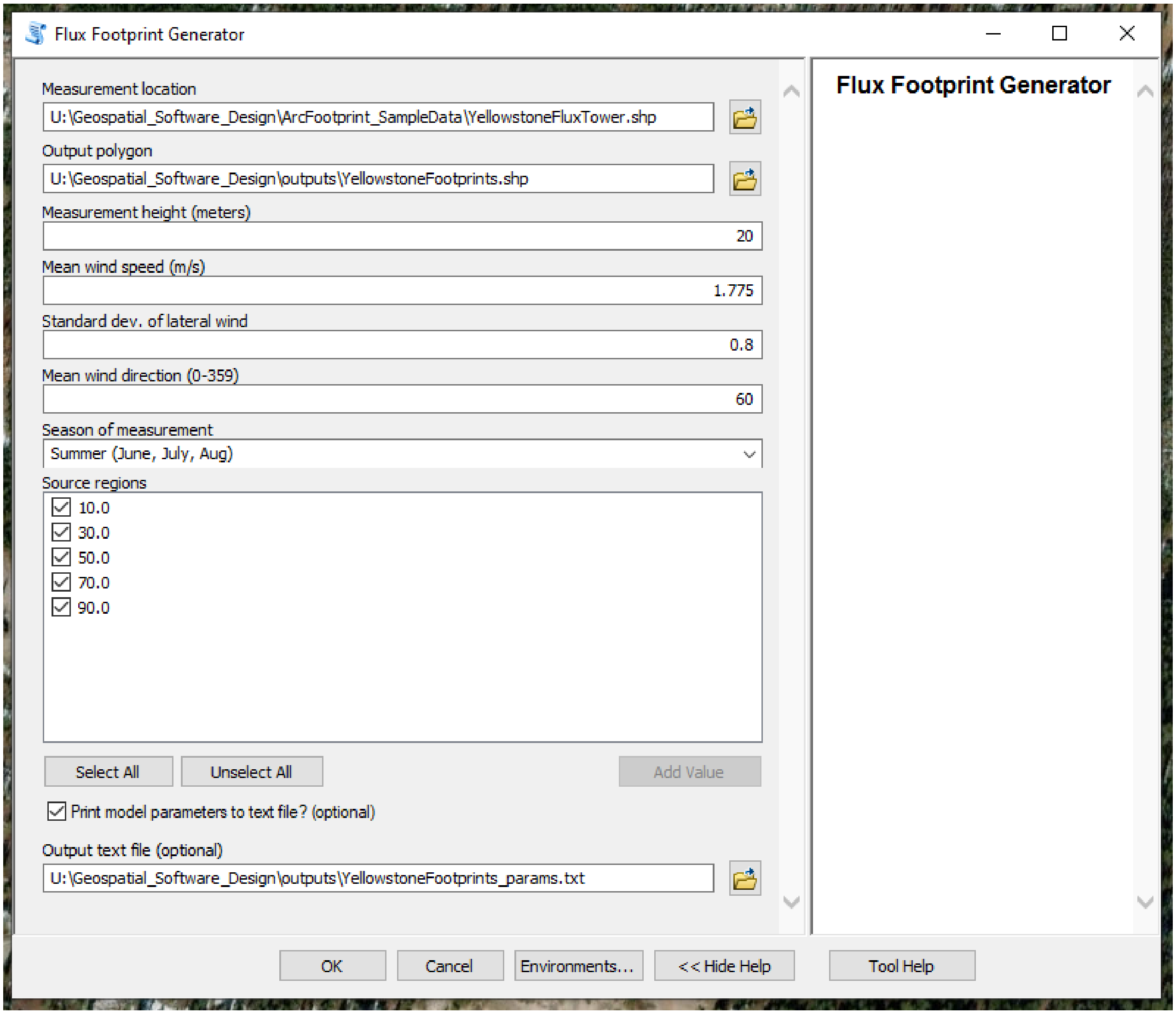


Demonstration 1: Yellowstone flux tower

Using sample flux tower data from Lewicki et al (2019)[6], footprints were estimated for one half-hour average.



Date	Time	Sensible heat flux	Latent heat flux	CO2 flux	H2O flux	Friction velocity	Monin-Obukhov length	Wind speed variance in cross-wind direction	Mean horizontal wind speed	Mean horizontal wind direction
	local time:	*watts per square meter*	*watts per square meter*	*micromoles*	*millimoles p*	*meters per second*	*meters*	*square meters per square second*	*meters per second*	*degrees from north*
5/12/2017	9:30	218.5	214.9	21.7	4.9	0.53	-48.18	1.47	4.27	171
5/12/2017	10:00	251.6	228.4	25.2	5.2	0.55	-47.88	1.67	4.94	167
5/12/2017	10:30	269.3	192.5	18.7	4.3	0.61	-61.15	1.66	5.37	169
5/12/2017	11:00	230.4	148.5	15.9	3.4	0.54	-47.45	1.77	4.95	171
5/12/2017	11:30	282.6	168.9	15.6	3.8	0.53	-37.71	2.02	4.99	175
5/12/2017	12:00	312.6	164	14.9	3.7	0.62	-54.7	2.33	5.5	179



Measurement location: U:\Geospatial_Software_Design\ArcFootprint_SampleData\YellowstoneFluxTower.shp

Output polygon: U:\Geospatial_Software_Design\outputs\YellowstoneFootprints.shp

Measurement height (meters): 20

Mean wind speed (m/s): 1.775

Standard dev. of lateral wind: 0.8

Mean wind direction (0-359): 60

Season of measurement: Summer (June, July, Aug)

Source regions:
- ☑ 10.0
- ☑ 30.0
- ☑ 50.0
- ☑ 70.0
- ☑ 90.0

☑ Print model parameters to text file? (optional)

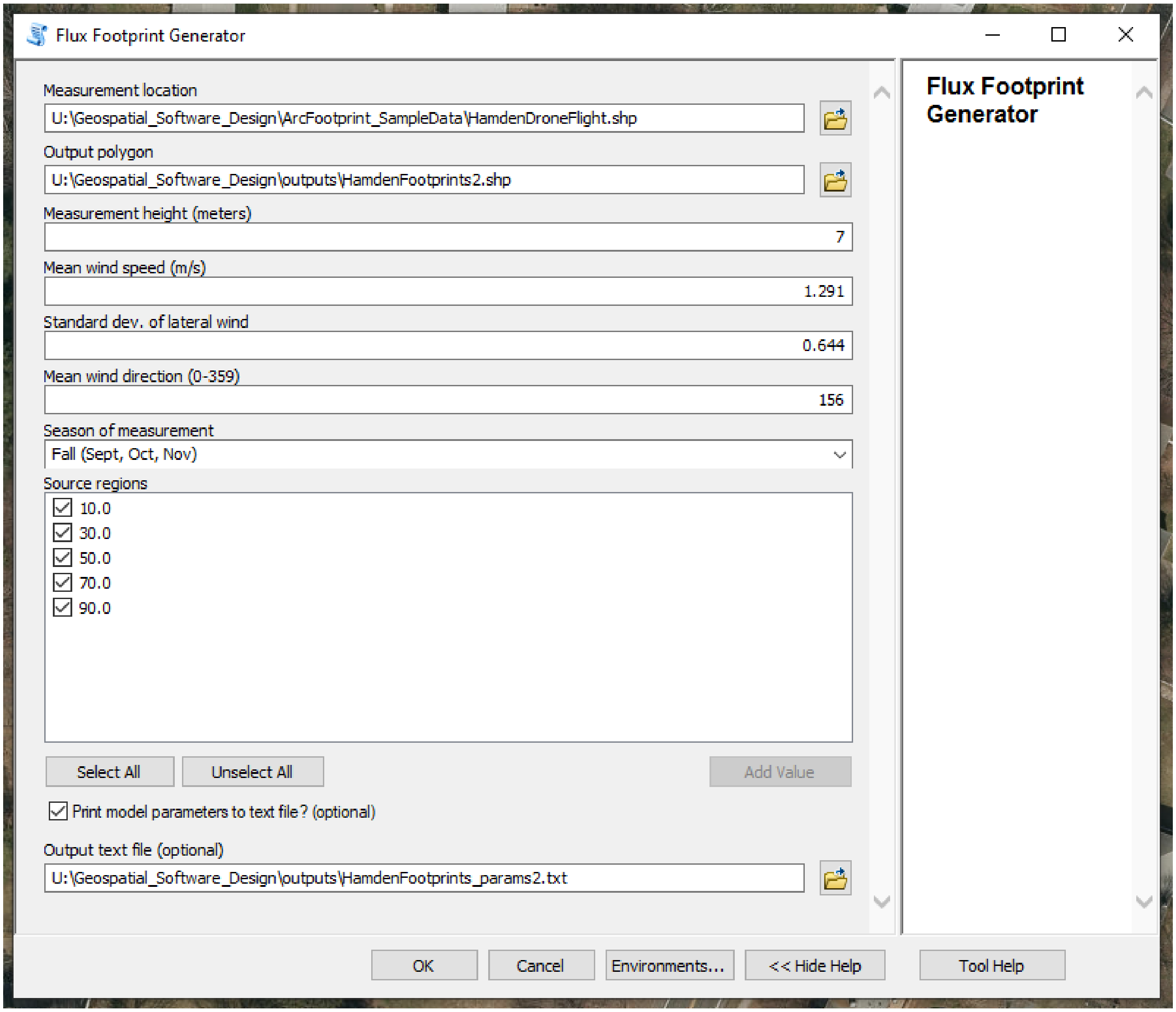
Output text file (optional): U:\Geospatial_Software_Design\outputs\YellowstoneFootprints_params.txt

---

[6] Lewicki, J.L., Kelly, P.J., and Clor, L.E., 2019, Gas and heat emission measurements at Solfatara Plateau Thermal Area, Yellowstone National Park (May-September 2017): U.S. Geological Survey data release, https://doi.org/10.5066/P9XOHUDO.

Demonstration 2: New Haven drone

Using data collected from my own field work[7], flying a UAV with an onboard anemometer, footprints were estimated from one 24-minute hovering flight.



Measurement location
U:\Geospatial_Software_Design\ArcFootprint_SampleData\HamdenDroneFlight.shp

Output polygon
U:\Geospatial_Software_Design\outputs\HamdenFootprints2.shp

Measurement height (meters)
7

Mean wind speed (m/s)
1.291

Standard dev. of lateral wind
0.644

Mean wind direction (0-359)
156

Season of measurement
Fall (Sept, Oct, Nov)

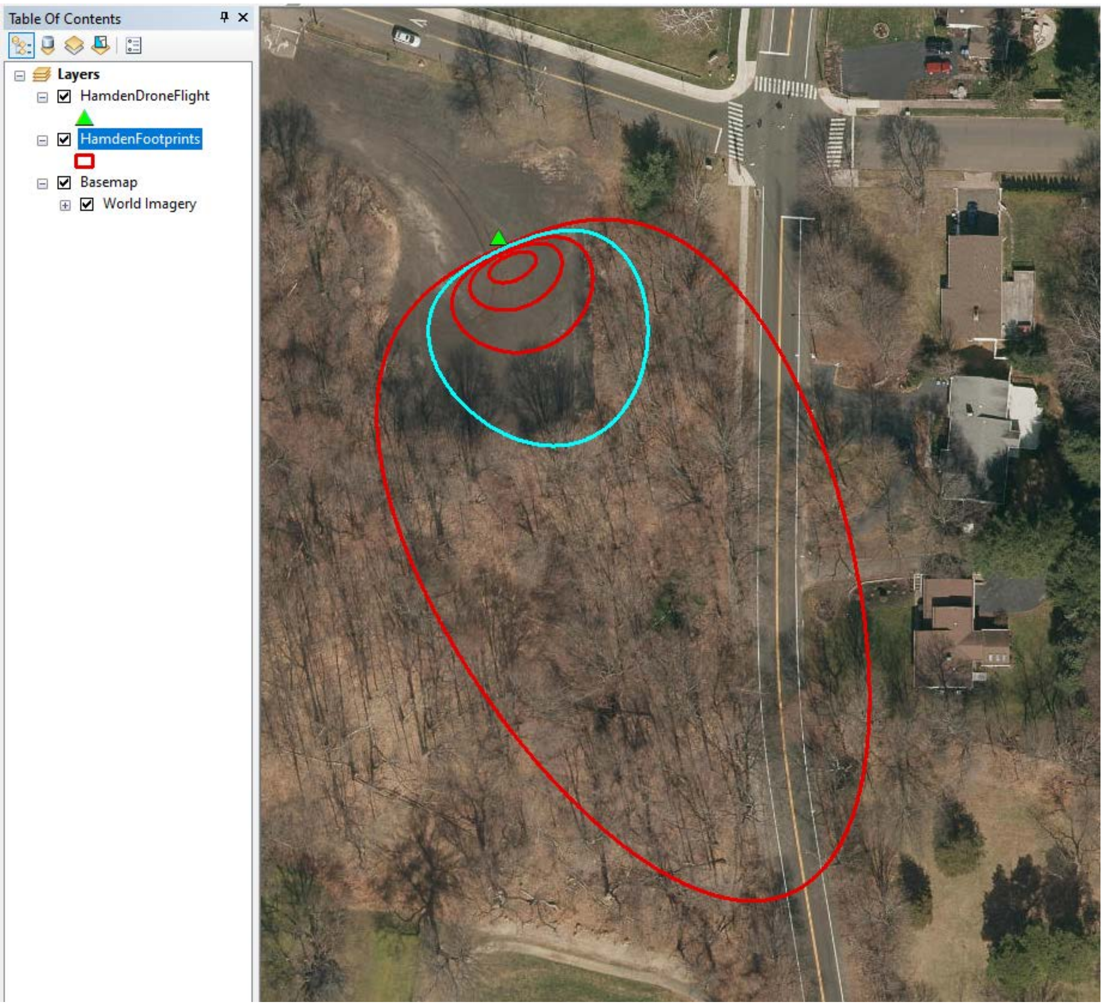
Source regions
☑ 10.0
☑ 30.0
☑ 50.0
☑ 70.0
☑ 90.0

☑ Print model parameters to text file? (optional)

Output text file (optional)
U:\Geospatial_Software_Design\outputs\HamdenFootprints_params2.txt

---

[7] O'Brien, M. and Schultz, N., unpublished data.



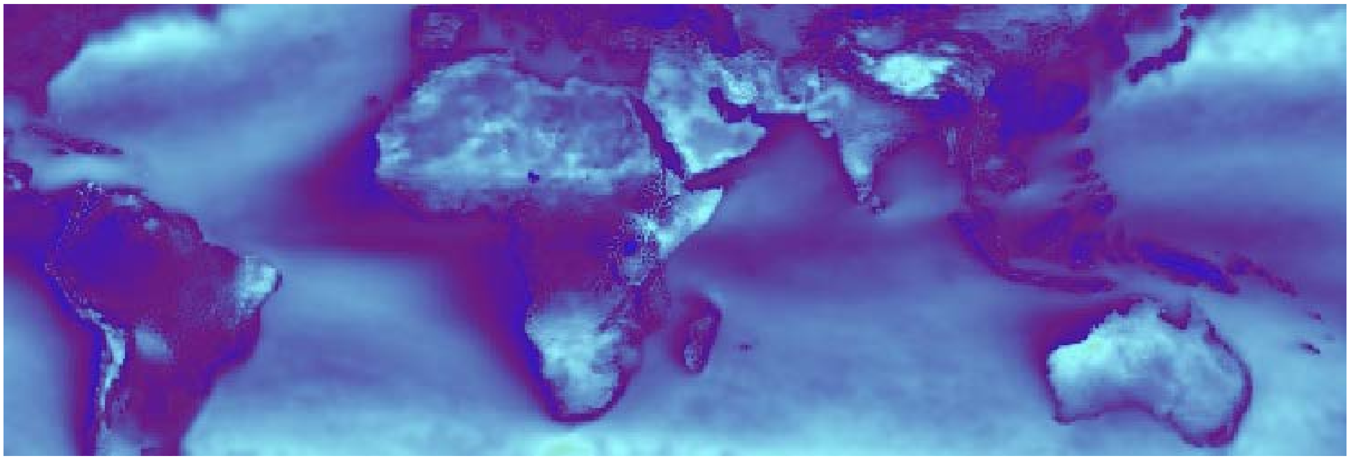
Output footprints from Yellowstone (top) and New Haven (bottom), each with one footprint selected.

Limitations and next steps

1) Very approximate BLH estimates within the Kljun model, and only for northern hemisphere.

The height of the atmospheric boundary layer (BLH) has a non-trivial impact on how far gases can travel and subsequently on flux footprints. BLH estimates presently in this tool[8] average over a lot of variation; the height of this atmospheric layer can change hundreds of meters over the course of a day due to temperature. In addition, the seasonality drop-down menu in my tool relies upon estimates from northern hemisphere seasons (hot June, cold January) whereas southern hemisphere sites would be reversed.

Next step: I experimented with creating a global-scale, course-resolution raster of BLH in Google Earth Engine, and averaging BLHs over 3 years of data (see image). When the user runs ArcFootprint, their sensor location could be compared to the raster of BLH values like a spatial look-up table, to find the suitable BLH for that season. However, I am not sure how to make the ArcFootprint tool "point" to this raster dataset without having the user add it as an input.



2) Output footprints have no metadata.

Next steps: Rather than clutter the user's computer with text logs tying footprints back to input parameters, I would prefer to write to the metadata of the footprint shapefile as it is created. ESRI's method for editing metadata programmatically involves writing and editing XML metadata documents, which can be accomplished in Python but may involve significant work.[9]

3) User must calculate wind variables (mean direction, SD of speed) themselves.

If a user has raw sensor data not yet averaged over a time interval (like a half hour), requiring the user to calculate their own mean and standard deviation statistics on their data increases opportunities for human error or misunderstanding tool instructions. Plus, it's an extra manual step, and who enjoys that?

Next steps: Add a widget in the tool dialog box that allows the user to identify a column in an attribute table (or CSV) containing wind data from their sensor. The statistics of mean wind direction, mean wind speed, and standard deviation of wind speed will be calculated automatically and inserted into the footprint estimation model.

---

[8] Current values estimated from Chu, Y., Li, J., Li, C., Tan, W., Su, T. and Li, J., 2019. Seasonal and diurnal variability of planetary boundary layer height in Beijing: Intercomparison between MPL and WRF results. *Atmospheric Research*, *227*, pp.1-13.

[9] See https://desktop.arcgis.com/en/arcmap/latest/manage-data/metadata/editing-metadata-for-many-arcgis-items.htm#ESRI_SECTION1_35F23B7429E8484890058C3C7A797D6E

Annotated script

```
"""
ArcFootprint, version 1.0
Created December 2019 by Mads O'Brien

This tool estimates a flux footprint based on a user-provided measurement location and
various meteorological observations.
The tool generates a shapefile that contains one or more polygon features, each
representing the area from which x% of a gas sensor's measurements originate.

Specify the following parameters in the ArcTool script:
DISPLAY NAME                DATA TYPE       DIRECTION
Measurement location        Shapefile       Input
Output polygon              Shapefile       Output
Measurement height          Double          Input
Mean wind speed             Double          Input
Mean wind direction         Long            Input           FILTER: 0 - 360
Season of measurement       String          Input           FILTER: Value list (see
'seasondictionary' variable)
Source regions              Double          Input
Print model parameters?     Boolean         Input           * optional
Output text file            Text file       Output          * optional


NOTE: the FFP() function in the following script is almost entirely written by Gerardo
Fratini and Natascha Kljun. I have borrowed excerpts of it within this tool.
Original metadata below:
    Derive a flux footprint estimate based on the simple parameterisation FFP
    See Kljun, N., P. Calanca, M.W. Rotach, H.P. Schmid, 2015:
    The simple two-dimensional parameterisation for Flux Footprint Predictions FFP.
    Geosci. Model Dev. 8, 3695-3713, doi:10.5194/gmd-8-3695-2015, for details.
    contact: n.kljun@swansea.ac.uk

    FFP Input
    zm     = Measurement height above displacement height (i.e. z-d) [m]
    z0     = Roughness length [m]; enter None if not known
    umean  = Mean wind speed at zm [m/s]; enter None if not known
             Either z0 or umean is required. If both are given,
             z0 is selected to calculate the footprint
    h      = Boundary layer height [m]
    ol     = Obukhov length [m]
    sigmav = standard deviation of lateral velocity fluctuations [ms-1]
        ustar  = friction velocity [ms-1]

    optional inputs:
    wind_dir = wind direction in degrees (of 360) for rotation of the footprint
    rs       = Percentage of source area for which to provide contours, must be between 10%
and 90%.
               Can be either a single value (e.g., "80") or a list of values (e.g., "[10,
20, 30]")
               Expressed either in percentages ("80") or as fractions of 1 ("0.8").
               Default is [10:10:80]. Set to "None" for no output of percentages
    nx       = Integer scalar defining the number of grid elements of the scaled footprint.
               Large nx results in higher spatial resolution and higher computing time.
               Default is 1000, nx must be >=600.
    rslayer  = Calculate footprint even if zm within roughness sublayer: set rslayer = 1
               Note that this only gives a rough estimate of the footprint as the model is
not
               valid within the roughness sublayer. Default is 0 (i.e. no footprint for
within RS).
               z0 is needed for estimation of the RS.
    crop     = Crop output area to size of the 80% footprint or the largest r given if
crop=1
```

The code block outlined in ORANGE is taken from Kljun et al; my additions are highlighted in YELLOW. The rest of the script is my own.

```
    fig      = Plot an example figure of the resulting footprint (on the screen): set fig = 
1.
               Default is 0 (i.e. no figure).

    FFP output
    x_ci_max = x location of footprint peak (distance from measurement) [m]
    x_ci  = x array of crosswind integrated footprint [m]
    f_ci  = array with footprint function values of crosswind integrated footprint [m-1]
    x_2d  = x-grid of 2-dimensional footprint [m], rotated if wind_dir is provided
    y_2d  = y-grid of 2-dimensional footprint [m], rotated if wind_dir is provided
    f_2d  = footprint function values of 2-dimensional footprint [m-2]
    rs       = percentage of footprint as in input, if provided
    fr       = footprint value at r, if r is provided
    xr       = x-array for contour line of r, if r is provided
    yr       = y-array for contour line of r, if r is provided
    flag_err = 0 if no error, 1 in case of error

    created: 15 April 2015 natascha kljun
    translated to python, December 2015 Gerardo Fratini, LI-COR Biosciences Inc.
    version: 1.3
    last change: 08/12/2017 natascha kljun
    Copyright (C) 2015,2016,2017,2018 Natascha Kljun

"""
###############################################################################
#%% Define FFP Function
def FFP(zm=None, z0=None, umean=None, h=None, ol=None, sigmav=None, ustar=None,
        wind_dir=None, rs=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8], rslayer=0,
        nx=1000, crop=False, fig=False):

    import numpy as np
    import sys
    import numbers

    #===========================================================================
    ## Input check
    flag_err = 0

    ## Check existence of required input pars
    if None in [zm, h, ol, sigmav, ustar] or (z0 is None and umean is None):
        raise_ffp_exception(1)

    # Define rslayer if not passed
    if rslayer == None: rslayer == 0

    # Define crop if not passed
    if crop == None: crop == 0

    # Define fig if not passed
    if fig == None: fig == 0

    # Check passed values
    if zm <= 0.: raise_ffp_exception(2)
    if z0 is not None and umean is None and z0 <= 0.: raise_ffp_exception(3)
    if h <= 10.: raise_ffp_exception(4)
    if zm > h: raise_ffp_exception(5)
    if z0 is not None and umean is None and zm <= 12.5*z0:
        if rslayer is 1: raise_ffp_exception(6)
        else: raise_ffp_exception(12)
    if float(zm)/ol <= -15.5: raise_ffp_exception(7)
    if sigmav <= 0: raise_ffp_exception(8)
    if ustar <= 0.1: raise_ffp_exception(9)
    if wind_dir is not None:
        if wind_dir> 360 or wind_dir < 0: raise_ffp_exception(10)
    if nx < 600: raise_ffp_exception(11)
```

```
    # Resolve ambiguity if both z0 and umean are passed (defaults to using z0)
    if None not in [z0, umean]: raise_ffp_exception(13)

    #===========================================================================
    # Handle rs
    if rs is not None:

        # Check that rs is a list, otherwise make it a list
        if isinstance(rs, numbers.Number):
            if 0.9 < rs <= 1 or 90 < rs <= 100: rs = 0.9
            rs = [rs]
        if not isinstance(rs, list): raise_ffp_exception(14)

        # If rs is passed as percentages, normalize to fractions of one
        if np.max(rs) >= 1: rs = [x/100. for x in rs]

        # Eliminate any values beyond 0.9 (90%) and inform user
        if np.max(rs) > 0.9:
            raise_ffp_exception(15)
            rs = [item for item in rs if item <= 0.9]

        # Sort levels in ascending order
        rs = list(np.sort(rs))


    #===========================================================================
    # Model parameters
    a = 1.4524
    b = -1.9914
    c = 1.4622
    d = 0.1359
    ac = 2.17
    bc = 1.66
    cc = 20.0

    xstar_end = 30
    oln = 5000 #limit to L for neutral scaling
    k = 0.4 #von Karman

    #===========================================================================
    # Scaled X* for crosswind integrated footprint
    xstar_ci_param = np.linspace(d, xstar_end, nx+2)
    xstar_ci_param = xstar_ci_param[1:]

    # Crosswind integrated scaled F*
    fstar_ci_param = a * (xstar_ci_param-d)**b * np.exp(-c/ (xstar_ci_param-d))
    ind_notnan     = ~np.isnan(fstar_ci_param)
    fstar_ci_param = fstar_ci_param[ind_notnan]
    xstar_ci_param = xstar_ci_param[ind_notnan]

    # Scaled sig_y*
    sigystar_param = ac * np.sqrt(bc * xstar_ci_param**2 / (1 + cc * xstar_ci_param))

    #===========================================================================
    # Real scale x and f_ci
    if z0 is not None:
        # Use z0
        if ol <= 0 or ol >= oln:
            xx  = (1 - 19.0 * zm/ol)**0.25
            psi_f = np.log((1 + xx**2) / 2.) + 2. * np.log((1 + xx) / 2.) - 2. *
np.arctan(xx) + np.pi/2
        elif ol > 0 and ol < oln:
            psi_f = -5.3 * zm / ol

        x = xstar_ci_param * zm / (1. - (zm / h)) * (np.log(zm / z0) - psi_f)
        if np.log(zm / z0) - psi_f > 0:
```

```
            x_ci = x
            f_ci = fstar_ci_param / zm * (1. - (zm / h)) / (np.log(zm / z0) - psi_f)
        else:
            x_ci_max, x_ci, f_ci, x_2d, y_2d, f_2d = None
            flag_err = 1
    else:
        # Use umean if z0 not available
        x = xstar_ci_param * zm / (1. - zm / h) * (umean / ustar * k)
        if umean / ustar > 0:
            x_ci = x
            f_ci = fstar_ci_param / zm * (1. - zm / h) / (umean / ustar * k)
        else:
            x_ci_max, x_ci, f_ci, x_2d, y_2d, f_2d = None
            flag_err = 1

    #Maximum location of influence (peak location)
    xstarmax = -c / b + d
    if z0 is not None:
        x_ci_max = xstarmax * zm / (1. - (zm / h)) * (np.log(zm / z0) - psi_f)
    else:
        x_ci_max = xstarmax * zm / (1. - (zm / h)) * (umean / ustar * k)

    #Real scale sig_y
    if abs(ol) > oln:
        ol = -1E6
    if ol <= 0:   #convective
        scale_const = 1E-5 * abs(zm / ol)**(-1) + 0.80
    elif ol > 0:  #stable
        scale_const = 1E-5 * abs(zm / ol)**(-1) + 0.55
    if scale_const > 1:
        scale_const = 1.0
    sigy = sigystar_param / scale_const * zm * sigmav / ustar
    sigy[sigy < 0] = np.nan

    #Real scale f(x,y)
    dx = x_ci[2] - x_ci[1]
    y_pos = np.arange(0, (len(x_ci) / 2.) * dx * 1.5, dx)
    #f_pos = np.full((len(f_ci), len(y_pos)), np.nan)
    f_pos = np.empty((len(f_ci), len(y_pos)))
    f_pos[:] = np.nan
    for ix in range(len(f_ci)):
        f_pos[ix,:] = f_ci[ix] * 1 / (np.sqrt(2 * np.pi) * sigy[ix]) * np.exp(-y_pos**2 / (
2 * sigy[ix]**2))

    #Complete footprint for negative y (symmetrical)
    y_neg = - np.fliplr(y_pos[None, :])[0]
    f_neg = np.fliplr(f_pos)
    y = np.concatenate((y_neg[0:-1], y_pos))
    f = np.concatenate((f_neg[:, :-1].T, f_pos.T)).T

    #Matrices for output
    x_2d = np.tile(x[:,None], (1,len(y)))  #creates new array
    y_2d = np.tile(y.T,(len(x),1))    #creates new array
    f_2d = f
    # f_2d is the array of distribution values -MAO


    #===========================================================================
    # Derive footprint ellipsoid incorporating R% of the flux, if requested,
    # starting at peak value.
    dy = dx
    if rs is not None:
        global xrs
        global yrs
        # xrs and yrs are the LISTS of COORDINATES for each flux footprint
```

```
        # declaring these variables as 'global' means that their values persist outside of
the FFP() function,
        # and I can manipulate them later using arcpy commands
        clevs = get_contour_levels(f_2d, dx, dy, rs)
        frs = [item[2] for item in clevs]
        xrs = []
        yrs = []
        for ix, fr in enumerate(frs):
            xr,yr = get_contour_vertices(x_2d, y_2d, f_2d, fr)  # x, y, f, lev
            if xr is None:
                frs[ix] = None
            xrs.append(xr)
            yrs.append(yr)

    else:
        if crop:
            rs_dummy = 0.8 #crop to 80%
            clevs = get_contour_levels(f_2d, dx, dy, rs_dummy)
            xrs = []
            yrs = []
            xrs,yrs = get_contour_vertices(x_2d, y_2d, f_2d, clevs[0][2])

    #===========================================================================
    # Crop domain and footprint to the largest rs value
    if crop:
        xrs_crop = [x for x in xrs if x is not None]
        yrs_crop = [x for x in yrs if x is not None]
        if rs is not None:
            dminx = np.floor(min(xrs_crop[-1]))
            dmaxx = np.ceil(max(xrs_crop[-1]))
            dminy = np.floor(min(yrs_crop[-1]))
            dmaxy = np.ceil(max(yrs_crop[-1]))
        else:
            dminx = np.floor(min(xrs_crop))
            dmaxx = np.ceil(max(xrs_crop))
            dminy = np.floor(min(yrs_crop))
            dmaxy = np.ceil(max(yrs_crop))
        jrange = np.where((y_2d[0] >= dminy) & (y_2d[0] <= dmaxy))[0]
        jrange = np.concatenate(([jrange[0]-1], jrange, [jrange[-1]+1]))
        jrange = jrange[np.where((jrange>=0) & (jrange<=y_2d.shape[0]-1))[0]]
        irange = np.where((x_2d[:,0] >= dminx) & (x_2d[:,0] <= dmaxx))[0]
        irange = np.concatenate(([irange[0]-1], irange, [irange[-1]+1]))
        irange = irange[np.where((irange>=0) & (irange<=x_2d.shape[1]-1))[0]]
        jrange = [[it] for it in jrange]
        x_2d = x_2d[irange,jrange]
        y_2d = y_2d[irange,jrange]
        f_2d = f_2d[irange,jrange]

    #===========================================================================
    #Rotate 3d footprint if requested
    if wind_dir is not None:
        wind_dir = wind_dir * np.pi / 180.
        dist = np.sqrt(x_2d**2 + y_2d**2)
        angle = np.arctan2(y_2d, x_2d)
        x_2d = dist * np.sin(wind_dir - angle)
        y_2d = dist * np.cos(wind_dir - angle)

        if rs is not None:
            for ix, r in enumerate(rs):
                xr_lev = np.array([x for x in xrs[ix] if x is not None])
                yr_lev = np.array([x for x in yrs[ix] if x is not None])
                dist = np.sqrt(xr_lev**2 + yr_lev**2)
                angle = np.arctan2(yr_lev,xr_lev)
                xr = dist * np.sin(wind_dir - angle)
                yr = dist * np.cos(wind_dir - angle)
                xrs[ix] = list(xr)
```

```
                yrs[ix] = list(yr)

    #===========================================================================
    # Fill output structure
    if rs is not None:
        return {'x_ci_max': x_ci_max, 'x_ci': x_ci, 'f_ci': f_ci,
                'x_2d': x_2d, 'y_2d': y_2d, 'f_2d': f_2d,
                'rs': rs, 'fr': frs, 'xr': xrs, 'yr': yrs, 'flag_err':flag_err}
    else:
        return {'x_ci_max': x_ci_max, 'x_ci': x_ci, 'f_ci': f_ci,
                'x_2d': x_2d, 'y_2d': y_2d, 'f_2d': f_2d, 'flag_err':flag_err}

#===============================================================================
#===============================================================================
def get_contour_levels(f, dx, dy, rs=None):
    '''Contour levels of f at percentages of f-integral given by rs'''
    import numpy as np
    from numpy import ma

    #Check input and resolve to default levels in needed
    if not isinstance(rs, (int, float, list)):
        rs = list(np.linspace(0.10, 0.90, 9))
    if isinstance(rs, (int, float)): rs = [rs]

    #Levels
    pclevs = np.empty(len(rs))
    pclevs[:] = np.nan
    ars = np.empty(len(rs))
    ars[:] = np.nan

    sf = np.sort(f, axis=None)[::-1]
    msf = ma.masked_array(sf, mask=(np.isnan(sf) | np.isinf(sf))) #Masked array for
handling potential nan

    csf = msf.cumsum().filled(np.nan)*dx*dy
    for ix, r in enumerate(rs):
        dcsf = np.abs(csf - r)
        pclevs[ix] = sf[np.nanargmin(dcsf)]
        ars[ix] = csf[np.nanargmin(dcsf)]

    return [(round(r, 3), ar, pclev) for r, ar, pclev in zip(rs, ars, pclevs)]

#===============================================================================
#===============================================================================
def get_contour_vertices(x, y, f, lev):
    import matplotlib._cntr as cntr
    c = cntr.Cntr(x, y, f)         # uses x_2d, y_2d, f_2d as inputs for drawing contours
    nlist = c.trace(lev, lev, 0)    # nlist is the raw, but nested, list of the coordinates
I want
    segs = nlist[:len(nlist)//2]   # the "//" divides two floats but truncates the
remainder
    N = len(segs[0][:, 0])
    xr = [segs[0][ix, 0] for ix in range(N)]
    yr = [segs[0][ix, 1] for ix in range(N)]

    return [xr, yr]  # x,y coords of contour points, removed from their nested list-in-a-
list

#===============================================================================
#===============================================================================
exTypes = {'message': 'Message',
           'alert': 'Alert',
           'error': 'Error',
           'fatal': 'Fatal error'}

exceptions = [
```

Important! The *matplotlib._cntr* function runs using matplotlib version 1.5.2 or earlier; I had to uninstall Anaconda from my machine in order to get the script to reference the correct ArcMap version. (This bug alone took up days/weeks of my time.)

```
    {'code': 1,
     'type': exTypes['fatal'],
     'msg': 'At least one required parameter is missing. Please enter all '
            'required inputs. Check documentation for
details.'},
    {'code': 2,
     'type': exTypes['fatal'],
     'msg': 'zm (measurement height) must be larger than
zero.'},
    {'code': 3,
     'type': exTypes['fatal'],
     'msg': 'z0 (roughness length) must be larger than
zero.'},
    {'code': 4,
     'type': exTypes['fatal'],
     'msg': 'h (BPL height) must be larger than 10m.'},
    {'code': 5,
     'type': exTypes['fatal'],
     'msg': 'zm (measurement height) must be smaller than h (PBL
height).'},
    {'code': 6,
     'type': exTypes['alert'],
     'msg': 'zm (measurement height) should be above the roughness sub-layer (12.5*z0).'},
    {'code': 7,
     'type': exTypes['fatal'],
     'msg': 'zm/ol (measurement height to Obukhov length ratio) must be equal or larger
than -15.5'},
    {'code': 8,
     'type': exTypes['fatal'],
     'msg': 'sigmav (standard deviation of crosswind) must be larger than zero'},
    {'code': 9,
     'type': exTypes['error'],
     'msg': 'ustar (friction velocity) must be >=0.1.'},
    {'code': 10,
     'type': exTypes['fatal'],
     'msg': 'wind_dir (wind direction) must be >=0 and <=360.'},
    {'code': 11,
     'type': exTypes['error'],
     'msg': 'nx must be >=600.'},
    {'code': 12,
     'type': exTypes['alert'],
     'msg': 'Using z0, ignoring umean.'},
    {'code': 13,
     'type': exTypes['error'],
     'msg': 'zm (measurement height) must be above roughness sub-layer (12.5*z0).'},
    {'code': 14,
     'type': exTypes['fatal'],
     'msg': 'if provided, rs must be in the form of a number or a list of numbers.'},
    {'code': 15,
     'type': exTypes['alert'],
     'msg': 'rs value(s) larger than 90% were found and eliminated.'},
        ]

def raise_ffp_exception(code):
    '''Raise exception or prints message according to specified code'''

    ex = [it for it in exceptions if it['code'] == code][0]
    string = ex['type'] + '(' + str(ex['code']).zfill(4) + '):\n '+ ex['msg']

    print('')
    if ex['type'] == exTypes['fatal']:
        string = string + '\n FFP_fixed_domain execution aborted.'
        raise Exception(string)
    else:
        print(string)
```

```
####################################################################################
####################################################################################

# Import necessary modules
import arcpy, traceback

try:

    ########## Define inputs and outputs ##########
    # erase any intermediate files in-memory from previous executions of this tool
    arcpy.Delete_management("in_memory")

    inputShape  = arcpy.GetParameterAsText(0)  # user-defined measurement location
    nameOfOutputShapefile  = arcpy.GetParameterAsText(1) # footprint shapefile to be
generated

    # Footprint estimation model parameters
    measureHeight = float(arcpy.GetParameterAsText(2))  # measurement height (m)
    meanWind = float(arcpy.GetParameterAsText(3))  # mean wind speed
    sdlatflux = float(arcpy.GetParameterAsText(4))  # stdev of lateral flux
    windrx = int(arcpy.GetParameterAsText(5)) # mean wind direction
    olength = -40.00   # very much an approximation... to be better constrained in future
versions

    season = arcpy.GetParameterAsText(6)# used to estimate height of the atmospheric
boundary layer
    seasondictionary = {"Summer (June, July, Aug)": 1000,
                        "Fall (Sept, Oct, Nov)": 600,
                        "Winter (Dec, Jan, Feb)": 300,
                        "Spring (March, April, May)": 1200,
                        "Unknown":775} # look up average values of boundary layer height
    PBLH = float(seasondictionary[season]) # returns the boundary layer height value as a
number

    sourceregions = arcpy.GetParameterAsText(7)   # user checks one or more "percent source
region" to generate
    valueList = [x.strip() for x in sourceregions.split(";")]  #splits single-string input
into a list of strings
    sourceList = [float(i) for i in valueList] # converts each item in the list from a
string to a float

    # Print model parameters to text file?
    ischecked = arcpy.GetParameterAsText(8)    # Boolean true or false
    resultLogFile = arcpy.GetParameterAsText(9)  # file path of output text file

    # text block to print model parameters to the ArcMap window
    modelParams = str("MODEL PARAMETERS USED: \n" +
                     "Measurement height: " + str(measureHeight) + " m \n" +
                     "Mean wind speed: " +str(meanWind) + " m/s \n" +
                     "Standard deviation of lateral wind: " +str(sdlatflux) + " m/s \n" +
                     "Mean wind direction: " +str(windrx) + " degrees \n" +
                     "Obukov length: " +str(olength) + "\n" +
                     "Season of measurement: " +str(season) + "\n" +
                     "Planetary boundary layer height: " +str(PBLH) + " m \n" +
                     "Source regions: " +str(sourceregions) + " percent \n")
    arcpy.AddMessage(modelParams)



    ########## Run FFP with user-defined parameters ##########
    FFP(zm=measureHeight, umean=meanWind, h=PBLH, ol=olength, sigmav=sdlatflux, ustar=0.53,
wind_dir=windrx, rs=sourceList)



    ########## Define X,Y coordinates of the input point-of-interest ##########
```

```
    # NOTE that input should only contain ONE point feature.
    # If not, script will only capture coordinates of last feature in the attribute table
    for rowx in arcpy.da.SearchCursor(inputShape, ["SHAPE@X"]):
        originx = rowx[0] # returns number as Double
    for rowy in arcpy.da.SearchCursor(inputShape, ["SHAPE@Y"]):
        originy = rowy[0]

    # Define CRS of the input point
    # **IMPORTANT** - Input MUST use a projected, planar coordinate system!
    spatial_ref = arcpy.Describe(inputShape).spatialReference

    # print information on input point to ArcMap window
    pointinfo= str("Measurement Location Coordinates X: "+str(originx)+" Y: "+str(originy)+
"\n" +
                   "CRS of input point: "+str(spatial_ref.name)+ "\n" +
                   "Linear unit: "+ str(spatial_ref.linearUnitName)+ "\n")
    arcpy.AddMessage(pointinfo)



    ########## Initialize list of XY coordinates **relative to a (0,0) origin** ##########
    # NOTE: list must be in correct order for eventual line-drawing.

    # Initialize 2 empty lists, a.k.a. containers to hold translated X and Y values
    newxshapes = []
    newyshapes = []

    # TRANSLATE each footprint vertex relative to the inputShape

    # If input point CRS uses meters, proceed with arithmetic
    if spatial_ref.metersPerUnit == 1.0:
        for boundary in xrs:  # for each requested footprint outline...
            boundxlist = []
            for vertex in boundary:   # for each vertex in a given footprint outline...
                newx = originx + vertex   # returns number as Double
                boundxlist.append(newx) # append translated X-value to list
            newxshapes.append(boundxlist)
        for boundary in yrs:  # for each requested footprint outline...
            boundylist = []
            for vertex in boundary: # for each vertex in a given footprint outline...
                newy = originy + vertex
                boundylist.append(newy) # append translated Y-value to list
            newyshapes.append(boundylist)
   # If the coordinate system uses a linear unit *other than meters*
    else:
        conversion = float(spatial_ref.metersPerUnit) # save a factor to convert other-
units to meters
        for boundary in xrs:  # for each requested footprint outline...
            boundxlist = []
            for vertex in boundary:   # for each vertex in a given footprint outline...
                newx = originx + (vertex / conversion)  # returns number as Double
                boundxlist.append(newx) # append translated X-value to list
            newxshapes.append(boundxlist)
        for boundary in yrs:  # for each requested footprint outline...
            boundylist = []
            for vertex in boundary: # for each vertex in a given footprint outline...
                newy = originy + (vertex / conversion)
                boundylist.append(newy) # append translated Y-value to list
            newyshapes.append(boundylist)

    # Initialize an empty list to contain coordinate *pairs* (X,Y) for each feature
    listoffeatures = []

    # zip the two lists of translated X and Y values into one list of tuples where each
tuple is a (X,Y) coordinate pair
    for i, j in zip(newxshapes, newyshapes):
```

```
        zippedfeature = zip(i,j)
        listoffeatures.append(zippedfeature)



    ######### Draw the footprint ellipses based upon coordinate pairs ###########
    # Initialize a list that will hold each of the Polyline objects, one for each footprint
    PolylineObjectList = []

    # Create a Polyline object for each requested footprint
    # Append to the list of Polyline objects
    for feature in listoffeatures:
        PolylineObjectList.append(arcpy.Polyline(arcpy.Array([arcpy.Point(*coords) for 
coords in feature]), spatial_ref))

    # turn each ellipse in my list of polylines into a polygon,
    # and save the polygon in memory with arbitrary filename
    arcpy.env.workspace = "in_memory"
    for i in range(len(PolylineObjectList)):
        arcpy.FeatureToPolygon_management(PolylineObjectList[i], 
"in_memory/polygon"+str(i))

    # Create list of each individual polygon footprint I generated, and merge into one 
shapefile
    polys = arcpy.ListFeatureClasses("polygon*")
    mergedpolys = "in_memory/mergedpolys"
    arcpy.Merge_management(polys, mergedpolys)



    ######### Edit attributes of final polygon output #########
    # Add attribute for each footprint's area, calculated from polygon's shape
    arcpy.AddField_management(mergedpolys, "footptAREA", "FLOAT")
    arcpy.CalculateField_management(mergedpolys, "footptAREA", "float(!SHAPE.area!)", 
"PYTHON")

    #SORT THE FEATURES FROM LARGEST TO SMALLEST (based on area), so smaller footprints draw 
on top of larger ones! (90, 70, etc)
    mergedsorted = "in_memory/mergedsorted"
    arcpy.Sort_management(mergedpolys, mergedsorted, [["footptAREA", "DESCENDING"]])

    # Add attribute "src_pct" to display the percent source region each polygon represents
    sourceListRev = sourceList[::-1]  # make reversed-order copy of sourceList, so it too 
uses a descending order (90, 70, etc)
    arcpy.AddField_management(mergedsorted, "src_pct", "LONG")
    pointer = 0  # used to iterate over the 'sourceListRev' list
    with arcpy.da.UpdateCursor(mergedsorted, "src_pct") as cursor:
        for row in cursor:
            row[0] = int(sourceListRev[pointer])  # assign each feature's "src_pct" 
attribute the corresponding value in sourceListRev
            pointer += 1
            cursor.updateRow(row)
    del row
    del cursor

    # Add attribute "obs_point" to record the name of the measurement location shapefile 
each footprint is based on (for record-keeping)
    arcpy.AddField_management(mergedsorted, "obs_point", "TEXT")
    rows = arcpy.UpdateCursor(mergedsorted)
    for row in rows:
        row.setValue("obs_point", str(inputShape))
        rows.updateRow(row)
    del row
    del rows

    # Save final footprint polygons, with attributes, to user-specified filepath
```

```
    arcpy.CopyFeatures_management(mergedsorted, nameOfOutputShapefile)
    arcpy.AddMessage("Output footprint shapefile: \t" + nameOfOutputShapefile + "\n")



    ######### Write model parameters to a text file, if user checked the box #########
    if str(ischecked) == 'true':
        import datetime
        systime = str(datetime.datetime.now())
        s = str(nameOfOutputShapefile+" footprint created at "+systime)  # system timestamp
when tool is executed
        f = open(resultLogFile,'a')
        f.write(modelParams)  # write the model parameters from beginning of script
        f.write(pointinfo)  # write information on measurement location from beginning of
script
        f.write(s)
        arcpy.AddMessage("Model parameters saved to text file.")

    # Delete intermediate files in temporary workspace
    arcpy.Delete_management("in_memory")

except Exception as e:
    # If unsuccessful, end gracefully by indicating why
    arcpy.AddError('\n' + "Script failed because: \t\t" + e.message )
    # ... and where
    exceptionreport = sys.exc_info()[2]
    fullermessage   = traceback.format_tb(exceptionreport)[0]
    arcpy.AddError("at this location: \n\n" + fullermessage + "\n")


####################################################################
```