Centralization and Automation of Kernel Density Estimation Calculations

Jennifer Carlson

Scott Kelley

Al Tafolla

Ellie Volosin

GPH 598: Geocomputation

Fall 2011

Final Project Report

**Motivation for and Objective of Project**

The objective of this project is to create a top level module to centralize the implementation of kernel density estimation (KDE) methods within the PySal library of spatial analysis tools. Currently, kernel density estimation methods are implemented in a variety of ways, both internally to PySal and additionally within other software programs associated with the The GeoDa Center for Geospatial Analysis and Computation, such as STARS. Centralizing KDE methods into one module will facilitate its implementation and is a practical objective, given that KDE is utilized for many purposes, such as exploratory point data analysis, point data smoothing and hot spot detection [de Smith, Goodchild, Longley, 2006], within a wide range of disciplines, including archeology, banking, climatology, hydrology, economics, genetics and physiology (Sheather, 2004).

Kernel density estimation is a way to estimate the density of data points using kernel methods (Sheather, 2004), and can be thought of as a smoothly curved surface that is fitted over each point in a data set. This curved surface is an estimation of the distribution of the data as a probability density function, and is non-parametrically derived from the data. ‘Kernel’ simply refers to the form of probability estimation function used to perform the estimation. The density at each data point is the sum of these estimated values derived from the kernel function at each point.

Kernel density estimation is more easily understood if related to a histogram, perhaps a more familiar method of data classification. In a histogram, data points are placed into discrete bins. However, such discrete placement of the data can mask some of its characteristics due to overgeneralization.

In contrast to histogram classification, kernel estimation reveals smoother transitions between data points. In kernel estimation, each data point is treated as if spread over a range, or neighborhood, where it is the center of its own neighborhood. These neighborhoods can be thought of as being analogous to histogram bins. The kernel estimator counts and sums the number of data points found within all the overlapping neighborhood layers to arrive at cumulative probability estimation, where more area under the curve corresponds to higher probabilities, similar to the way height in a histogram bin corresponds to more values within the bin. The difference here is that in effect, each data point is weighted by how far it is from the center of all the other overlapping neighborhoods and so is not restricted by the discrete values found at the bounded edges of a histogram bin. Values are highest where neighborhoods overlap the most but smoothly diminish at points where there is less overlap. These cumulative sums as estimated by the kernel are then rescaled by the range used to construct the neighborhoods so that the area under all curves sums to one, to assure a standardized view of the relationships between curves. [sensu de Smith, Goodchild, Longley, 2006; figure out how to cite this http://www.math.caltech.edu/~alberts/talks/KernelEstimation.pdf] The result is an interpolation over distances between points, yielding something similar to a risk surface.

**Considerations for implementation**

General form for kernel density estimation is:

\begin{displaymath}
\hat{{f}}(x)=\frac{1}{n}\sum_{i=1}^{n}K\left(\frac{x-x(i)}{h}\right)\end{displaymath}

K in the equation refers to the form of the kernel, or form of probability function. Kernels used are Gaussian (or the standard normal distribution), Quartic, Exponential, Triangular, Uniform and Epanechnikov (see Figure 1).

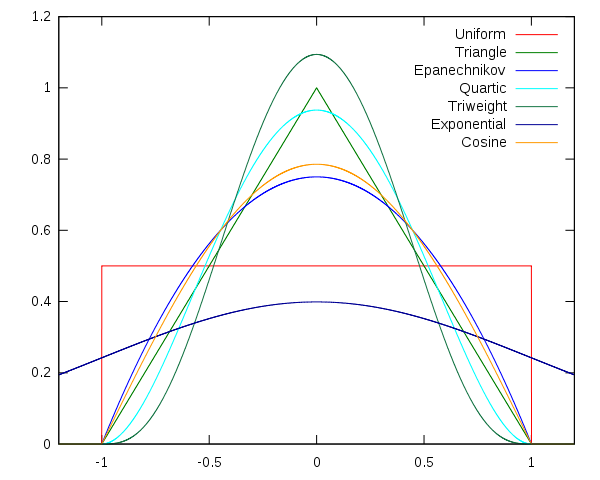


Figure 1. Kernel Shapes. *Source: (Li and Racine 2007)*

The shape of these functions determines, in part, the contribution each point will make to the total probability estimation. This contribution is also determined by the numerator, whereby x – x(i) defines the range, or spread of influence of the data point x(i). In the denominator, the value for h, or bandwidth, is used to modify the range for x(i).

The kernel estimate depends less on the shape chosen for K than on the value of its bandwidth h (de Smith, Goodchild, Longley, 2006; Sheather, 2004). Larger values for h decrease the difference between x and x(i), smoothing more because the kernel will be estimated over smaller distances. Large values for h over-smooth the estimation, while too small of values for h under-smooth.

**Kernel Density Estimation Functions**

A number of functions are commonly used to perform kernel density estimation (KDE). Particular needs and experience will dictate which function to use, though bandwidth selection is often more important than kernel selection. These functions are shown in Figure 1. The Gaussian, triangular, and uniform kernel functions have been included in this Kernel Density Estimation Module. Each particular equation used in KDE typically consists of the kernel and the bandwidth, also known as the smoothing parameter. The KDE module requires as input a list of data points (occurrences), each of which the module sets to the variable “mu” in turn. The bandwidth is also required and is set to the variable “sig”. The code automatically determines grid values for which a value is to be determined. Each of these grid values is set the variable “xi” in its proper turn.

The Gaussian Kernel is among the more commonly used. It diffuses spreads the point value of an occurrence by assuming a normal distribution about the occurrence point. The one-dimensional Gaussian kernel function is of the form:

Here, “” represents the distance between the data point and the point for which a value is to be calculated. In the KDE module, “” is calculated as “”. The normal distribution, and hence, the Gaussian function are unbounded, extending to infinity and returning a non-zero value for every point to infinity, though they grow negligibly small beyond the bandwidth. The bandwidth “” represents one standard deviation of the function. The term in the equation is the normalization constant, which “normalizes” the function so that the integral over each is equal to 1. Thus, increasing the bandwidth reduces the amplitude of the function, increasing the smoothing effect of the kernel substantially.

The triangular kernel function produces a simple linear decrease in value as distance increases from a data point. In the simple case where the value at a data point is 1, and the bandwidth is also 1, the triangular kernel takes the form:

(*note that bandwidth)*

However, in order to accommodate a variable bandwidth, important for our code if a user wants to test multiple bandwidths, the kernel takes the form:

The value of a point within the bandwidth in a triangular kernel function decreases at a rate equal to the slope of 1/bandwidth, and is zero beyond the bandwidth.

The third KDE function included in the KDE module is the uniform function. This is a very simple function that does not decrease the value of point within the bandwidth with increasing distance. Instead, the uniform function simply assigns a value equal to the data point to every point within the bandwidth. Values then drop abruptly to zero beyond the bandwidth, resulting in a very pronounced edge effect. The uniform kernel function takes the form:

The value used for in the KDE module is 1. Therefore, using the uniform kernel function would be equivalent to laying a disk of 1-unit thickness and radius equal to the bandwidth over the data point.

**Code Workflow**

While kernels can be calculated for any number of dimensions, this module generates 2-dimensional kernel outputs. Many geographic events are represented in a 2-dimensional Cartesian space, where events are represented with x,y values. Examples include locations of weather stations, locations of crime incidents, and parcel centroids. A kernel calculation will “smooth” incidents of these phenomena over a selected distance (bandwidth) away from the represented point, creating a interpolation mechanism to measure the probability of an event Z happening at x1,y1, based on its proximity to all x,y events. To accommodate this, the module looks for the following three input arguments in order to run:

* Points (represented in x,y values in a list)
* Bandwidth
* Resolution

Detailed documentation of all Kernel module functions and overall module workflow, along with examples of input and output, can be found in Appendix A, but the following section offers a general overview.

Points must be passed into the module with the format [x1,y1,x2,y2,…,xn,yn] in order for the x- and y- values to be evaluated separately, important for use in later functions (Appendix A). The bandwidth and resolution arguments must be numerical in nature.

The Kernel() class instantiates with the points (and the derivative x- and y-value lists), bandwidth, and resolution values stored at this top level. Figure 2 shows the generalized workflow of the module’s logic.

The first function is the Preplists() function, documented in Appendix A, which splits the input points into x and y-lists, set as xvals and yvals. The outputgrid\_x, and outputgrid\_y values– generated from the (input points + bandwidth) / resolution – are stored for use in later functions.

Resolution is important to the code’s output, as that input directly affects the output grid values. In the Preplists() function, the minimum bounding box of the input points - plus a buffer equal to the bandwidth - is divided into equal parts by the resolution value, creating an output grid for both the x- and y-dimensions. Where these x- and y- values intersect, the kernel heights will be reported, creating an output matrix of [outputgrid\_x1, outputgrid\_y1, z, outputgrid\_x1, outputgrid\_y2, z,… outputgrid\_xn, outputgrid\_yn, z], where ‘z’ represents the kernel height. See Appendix A for more details. The resolution selection is user-dependent: the larger an output matrix desired, the lower the resolution should be, and vice versa.

Figure 2. Conceptualization of Kernel Module workflow

The next function is the calculate() function, where the blueprint for actual kernel height derivations take place, based on either the Gaussian, Triangular, and Uniform functions within the module. They require arguments ‘xi’, ‘mu’, and ‘sigma’, along with ‘method’ which will be provided by the individual kernel shape functions called elsewhere. The function is formed on a conditional foundation, where 1 == Gaussian, 2 == Triangular, and 3==Uniform. These conditions – the ‘method’ argument – are provided by the individual kernel shape functions called elsewhere. The user will not actually see or interact with the calculate function, but it operates in the background when any of derivative functions are called later.

For the final portion of the module, the user must choose to receive a kernel height value in one of two formats: 1) the output grid generated in the Preplists() function, or 2) a single point of the user’s choice, provided this point is within an area that could be impacted by a kernel smoothing function (Figure 3). If the former is desired, enter the gaussian(), triangular(), or uniform() functions. If the latter is desired, pass an x-coordinate and y-coordinate into the gaussian\_point(), triangular\_point(), and uniform\_point() functions to receive the kernel height at that location.

The ‘xi’, ‘mu’, and ‘sig’ arguments from the calculate() function will correspond to the xvals/yvals, outputgrid\_x/outputgrid\_y, and bandwidth values, respectively (Appendix A). These are used by the gaussian(), triangular(), and uniform() functions, along with the gaussian\_point(), triangular\_point(), and uniform\_point() functions. Each of these functions sum the cumulative heights of all kernel effects within the area of interest when reporting a final kernel height.

The gridding function uses two loops: one that cycled through the output grid points, and an inner loop that cycles through the input points. The single point function contains one loop, which only loops through the input observation points as an output grid is unnecessary.

The kernel height of the x- and y- values (in either the outputgrid or single point kernel calculations) are evaluated separately, then multiplied together to generate a 2-dimensional single kernel height at each requested Cartesian location. Depending on which type of output function format the user chose, a matrix of [outputgrid\_x1, outputgrid\_y1, z, outputgrid\_x1, outputgrid\_y2, z,… outputgrid\_xn, outputgrid\_yn, z] or a single list of [x,y,z] will be returned to the user. The kernel heights will vary based on the kernel shape function chosen.

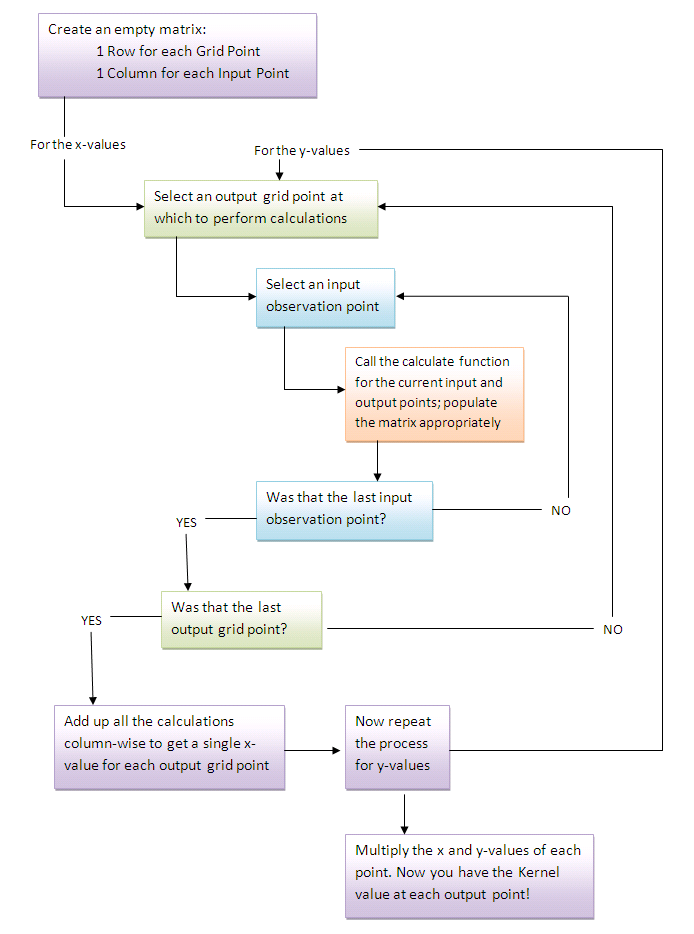


Figure 3. Flowchart of \_point() modules’ code.

**Future work**

This module represents the first step in centralizing kernel density estimation calculations. Gaussian, triangular, and uniform kernel functions are available to the user at the moment, but Epanechnikov, Quadratic, and Quartic functions are not. These additional calculations can be included in the module in the future, using the structure provided. Through extra conditional statements in the current ‘calculate()’ function, and the addition of functions using the ‘gaussian()’, and ‘gaussian\_point()’, etc. blueprint, these other kernel functions can assimilate into the existing code with ease.

In its current form, the code allows the user to view kernel output values at a grid defined by the maximum and minimum x,y values (with bandwidth buffer added), divided into equally-sized grid cells determined by a resolution value. In addition, a user can query a single specific location’s kernel height. Creating a variable-sized output matrix would be a useful addition, along with the ability to query multiple locations within the area of interest. Another useful variation on the output grid functionality is the matrix output form of [x1,y1, z, x2,y2,z….x2,y2,z]. We chose not to overwhelm the user with a matrix of this size in the case of a large coverage area, but this could certainly be implemented for users interested in a large output matrix. This would be particularly useful in the case of a raster dataset creation for geovisualization.

Though the kernel module is not designed for direct visualization use, it would be helpful to see these outputs in a graphic interface built to display geographic data. Finally, this module represents the knowledge of our group’s efforts to understand and automate kernel density estimation calculations. Code optimization by those more experienced in both Python and kernel density estimation calculations could improve the performance on this model. We tested this code on relatively small datasets, so its computational performance on a very large input dataset is yet to be determined.

**References**

|  |
| --- |
|  |

deSmith, Goodchild, and Longeley. 2001. Geospatial Analysis - a comprehensive guide. 3rd edition . <http://www.spatialanalysisonline.com/output/>

Sheather, Simon. 2004. Density Estimation. *Statistical Science*: 19 (4), 588-597

Liq, Qi and Jeffrey Racine. 2007. Nonparametric Econometrics: Theory and Practice. Princeton University Press.

APPENDIX A – CODE DOCUMENTATION

Documentation template prepared by Sphinx, populated by Kernel project team:

Welcome to Kernel's documentation!  
==================================  
  
Contents:  
  
.. toctree::  
 :maxdepth: 2  
  
.. module:: Kernel  
.. class:: Kernel(points[list], bandwidth, resolution)  
 :synopsis: Describes the functions housed within Kernel class  
  
This class houses the centralized functions for KDE. Included are functions that take an input list in the following format: [x1,y1,x2,y2,...,xn,yn] and run a kernel density estimation calculation over these input points.  
  
The following variables are instantiated and stored in the \_\_init\_\_function: these are the input points, the x-coordinates and y-coordinates of the input data, the bandwidth, and the resolution of the output coordinates in question. Defaults are provided, but can be edited by the user.  
  
The user must input the following when calling this module:  
  
A) Points: a list of points at which observations are made in the format [x1, y1, x2, y2, ..... xn, yn] where n is the total number of observations.  
B) Bandwidth: corresponds to the distance over which an observed point affects the grid points around it  
C) Resolution: This directly corresponds to the number of grid points at which the user would like to see the output.  
  
"""  
Example of inputs needed:  
 points = [11, 22, 23, 33, 33, 44, 45, 55]  
 b = 2  
 r = 10  
Will be needed to call:  
 k = Kernel(points, b, r)  
  
.. function:: PrepLists()  
Preplists take the input list from the user and splits the x- and y- values into separate lists. Since coordinate pairs create a 2-dimensional output, the x- and y-values will need to be evaluated separately, then combined again near the final step of the kernel calculation.  
  
After this step, the function finds the minimum bounding coordinates of the coverage area, then adds the bandwidth distance as a buffer. Within this region, effects from the kernel smoothing function are possible.  
  
This sets up the output grid functionality. The maximum and minimum range values are divided into equal parts by the input resolution value. This creates an x,y grid, where output kernel values will be returned in the calculation functions.  
  
Users also have the option - in later functions - to access a kernel value at a specific point, so long as that point falls within the bounding coordinates of the output grid.  
  
Values returned from this function are the xvals, yvals, outputx\_grid, and outputy\_grid.  
  
-----  
Example:  
 points = [11, 22, 23, 33, 33, 44, 45, 55]  
 b = 2  
 r = 10  
  
 k = Kernel(points, b, r)  
 k.preplists()  
-----  
 This k.preplists() command would create an input xvals of [11,23,33,45] and an input yvals of [22,33,44,55]. In xy space, this would correspond to [(11,22), (23,33), (33,44), and (45,55)]   
 The bandwidth above is '2', so the output x grid would have bounding coordinates of 9 (11-2), and 47 (45+2). The output y grid would have bounding coordinates of 20 (22-2) and 57 (55+2). Within this minimum bounding rectangle of (9,20) and (47,57) would contain any possible results of any following kernel function. The x- and y- grids would be parsed into 10 equal parts, per the resolution, and these become the outputx and outputy lists. These values are returned from the Preplists function to pass into the kernel calculation functions, along with the xvals and yvals variables.  
  
.. function:: calculate(xi, mu, sig, method)  
  
 The calculate function offers three kernel density estimation models: 1) Gaussian (normal distribution), 2) Triangular, and 3) Uniform kernels. Each conditional statement in the function will calculate a kernel height differently, but each will use: "xi", which is the value for each location from the outputgrid\_x or outputgrid\_y lists: "mu" which corresponds to each location from the xvals and yvals lists: and "sig", which is the bandwidth value. The 'method' argument is a numerical key, accessed by later functions to alert the calculate() function to the appropriate condition.  
  
.. function:: gaussian()  
 This function applies a normal distribution smoothing function around each xlist or ylist location, then locates each requested x- and y- location in the output grids, and calculates the cumulative height of all gaussian kernel heights at each output grid coordinate pair. Total height will be the sum of the number of input points' effect, expressed in the 'rows' variable.  
  
 The x- and y- values are evaluated separately, then multiplied to generate a Gaussian kernel height at each output grid location. Returned from this function is 'r': each coordinate pair in the output grid and its associated kernel height, which is the 'gaus\_kernel' variable.  
  
.. function:: gaussian\_point(xpoint, ypoint)  
 A user must input a desired x- and y- coordinate to call this function, and that point must be within the output grid area in order to impacted by kernel smoothing functions. If not the case, the function prompts the user to re-enter a valid coordinate pair.  
 If passed, then this function works exactly like the gaussian() function, but uses the specific x- and y- points instead of the output grid x- and y- coordinates. Returned from this function is 'r': the requested coordinate pair and its kernel height, which is the 'gaus\_kernel' variable.  
  
.. function:: triangular()  
 This function applies a triangular smoothing function around each xlist or ylist location, then locates each requested x- and y- location in the output grids, and calculates the cumulative height of all gaussian kernel heights that originate at all input points over each output grid coordinate pair. Total height will be the sum of the number of input points' effect, expressed in the 'rows' variable.  
  
 The x- and y- values are evaluated separately, then multiplied to generate a Triangular kernel height at each output grid location. Returned from this function is 'r': each coordinate pair in the output grid and its associated kernel height, which is the 'tri\_kernel' variable.  
  
.. function:: triangular\_point(xpoint, ypoint)  
 A user must input a desired x- and y- coordinate to call this function, and that point must be within the output grid area in order to impacted by kernel smoothing functions. If not the case, the function prompts the user to re-enter a valid coordinate pair.  
 If passed, then this function works exactly like the triangular() function, but uses the specific x- and y- points instead of the output grid x- and y- coordinates. Returned from this function is 'r': the requested coordinate pair and its kernel height, which is the 'tri\_kernel' variable.  
  
.. function:: uniform()  
 This function applies a uniform distribution smoothing function around each xlist or ylist location, then locates each requested x- and y- location in the output grids, and calculates the cumulative height of all uniform kernel heights that originate at all input points over each output grid coordinate pair. Total height will be the sum of the number of input points' effect, expressed in the 'rows' variable.  
  
 The x- and y- values are evaluated separately, then multiplied to generate a uniform kernel height at each output grid location. Returned from this function is 'r': each coordinate pair in the output grid and its associated kernel height, which is 'uni\_kernel'.  
  
.. function:: uniform\_point(xpoint, ypoint)  
 A user must input a desired x- and y- coordinate to call this function, and that point must be within the output grid area in order to impacted by kernel smoothing functions. If not the case, the function prompts the user to re-enter a valid coordinate pair.  
 If passed, then this function works exactly like the uniform() function, but uses the specific x- and y- points instead of the output grid x- and y- coordinates. Returned from this function is 'r': the requested coordinate pair and its kernel height, which is 'uni\_kernel'.  
  
Building on the example from earlier, calling the Gaussian function:  
  
 points = [11, 22, 23, 33, 33, 44, 45, 55]  
 b = 2  
 r = 10  
  
 k = Kernel(points, b, r)  
 k.preplists()  
  
 print 'The x-coordinate, y-coordinate, and gaussian kernel values are:', k.gaussian()  
  
Will return:  
  
The x-coordinate, y-coordinate, and gaussian kernel values are: [(9.0, 20.0, 0.01463745789778124), (13.222222222222221, 24.111111111111111, 0.012296353589811163), (17.444444444444443, 28.222222222222221, 6.9579895793571664e-05), (21.666666666666668, 32.333333333333329, 0.030138661108901638), (25.888888888888889, 36.444444444444443, 0.0032089432747310163), (30.111111111111111, 40.555555555555557, 0.0032089432747310163), (34.333333333333336, 44.666666666666664, 0.030138661108901635), (38.555555555555557, 48.777777777777771, 6.9579895793572098e-05), (42.777777777777779, 52.888888888888886, 0.01229635358981114), (47.0, 57.0, 0.01463745789778124)]