The sampSurf Package Overview

Jeffrey H. Gove*
Research Forester
USDA Forest Service
Northern Research Station
271 Mast Road
Durham, New Hampshire 03824 USA
e-mail: jgove@fs.fed.us or e-mail: jhgove@unh.edu

Monday 12th December, 2011
4:31pm

Contents

1 Introduction

The `sampSurf` package is concerned with generating sampling surfaces as an aid to learning about new or existing areal sampling methods in natural resources. The sampling surface concept was first proposed by Williams (2001a,b) as a method for visualizing the variability in sampling methods and provides an intuitive way to make comparisons among methods. It can be thought of as a discrete approximation of the continuous or infinite population paradigm and hence is related to the Monte Carlo exposition in Valentine et al. (2001). Additionally, it has been applied in a number of studies in addition to those already mentioned, such as Williams and Gove (2003), Gove et al. (2005), Gove and Van Deusen (2011) and Gove et al. (2012).

*Phone: (603) 868-7667; Fax: (603) 868-7604.

In the sampling surface approach, we map an area—a tract—into discrete grid cells of some desired resolution. Then we take a population of standing trees or down logs (hereafter simply "stems") whose locations are mapped within the area and apply an areal sampling method to them. Areal sampling methods, when combined with a stem, form an inclusion zone of known area and perimeter surrounding the stem[1]. The inclusion zone is simply the area within which a sample point could fall and select (include) the stem into the sample. This is true regardless of the areal sampling method. We coupled the sampling method to the stem because often the size of the inclusion zone varies with some dimension of the stem—the essence of probability proportional to size (PPS) sampling. For example, with Bitterlich sampling of standing trees, the circular plot inclusion zone size is proportional to the tree basal area (or breast height diameter squared). Even with fixed-area plots, where the size of the plot does not depend on any stem attribute, it is still convenient to think of the inclusion zone as couple to the stem. To generate a sampling surface then, all of the grid cells within the inclusion zone of each stem are assigned the per unit area estimates associated with the sampling method for that stem; these might include cubic and board foot volumes, cross-sectional areas, biomass, and the like. When inclusion zones for different elements in the population overlap, the per unit area estimates sum for the intersected grid cells. In this way, we build up a "sampling surface" individually for each quantity to be estimated. The smaller the grid cell size, the more accurate the approximation of the surface, and also the more time it takes to generate the full surface. One can now see that this approach is like a Riemann sum approaching an integral in one or more dimensions. So just as we expect the Riemann sum approximation to converge to the integral as the partitions get smaller, so we also expect the same of the sampling surface approximation to the true continuous surface as the grid cell resolution increases (cell sizes get smaller).

The `sampSurf` package automates the above procedure in several steps. It does this using S4 style classes and methods, which allow strict validity checking of objects. But more importantly, as a true object-oriented system, S4 has a full inheritance, which allows one to take the full functionality of the classes and methods that have been written and expand them by creating new classes for more detailed objects.

The package handles the more important fixed-area plot sampling methods for down logs, plus several other sampling methods for down wood such as point relascope (Gove et al., 1999), distance limited (Gove et al., 2012), and perpendicular distance methods (Williams and Gove, 2003; Williams et al., 2005; Ducey et al., 2008). Methods for standing trees include fixed-area circular plot and horizontal point (prism) sampling. Other sampling methods can be added to the foundational classes and methods (for different generic functions) that have been established without a great deal of programming. Finally, even though it may be more reasonable to think of applying these tools to simulate synthetic populations of individual stems, the facility exists for using existing data taken from an inventory where one of the supported areal sampling methods was used. The *Extending the `sampSurf` Package* vignette shows how to extend the base functionality of `sampSurf` through the S4 paradigm to create routines to handle new sampling methods.

[1] In most cases this is true, but there are exceptions: the "standup" method has inclusion zone surrounding only the butt of the log, for example.

2 sampSurf Package Classes

In this section we introduce the major `sampSurf` classes in a progressive way, building towards generating a sampling surface. Following that, an example is provided showing the process from start to finish. Many more details are provided in the individual vignettes associated with each class in the package documentation where, for example, different class constructors are illustrated.

2.1 The "Stem" class

We refer to all trees and down logs as "stems" above for a good reason. In this class, both trees and down logs are envisioned as being subclasses of an overall conceptual "Stem" class. In R parlance, the "Stem" class is a virtual class, and specifies all of the attributes that a down log and standing tree might have in common, such as species. Other attributes that are not shared, are defined in the individual subclass specifications themselves, while inheriting the functionality of the superclass, in this case "Stem". For example, down logs have small- and large-end diameters (well so do standing trees, but they are rarely measured), while standing trees have diameters at breast height. Therefore, the subclasses for down logs and standing trees will be somewhat different, while still inheriting the common "Stem" characteristics from that superclass.

2.1.1 The "downLog" subclass

This subclass is for down logs only. It contains the basic information about the logs, such as diameters and length. It also allows the generation of taper curves for the log if none are available from measurements. These are generated from a standard taper equation that can model many different shapes from neiloid to paraboloid (Gove and Van Deusen, 2011). In addition, the geometrical information is stored for the log so that it can be represented on a two-dimensional surface. The log's center location and a polygonal profile outline are stored in the class slots for the object, using spatial polygon classes from the `sp` package. One could easily envision extending this class and its constructor to handle other taper equations for example through inheritance. In fact, this was a decision in the design of the class, rather than have the taper equation itself stored as a component in the object, for example. (See *The "Stem" Class* vignette for more information.)

2.1.2 The "standingTree" subclass

Similar to the "downLog" class, this subclass contains the basic information for standing trees. It uses the same default taper equation and spatial representation, but displays the tree diameter at breast height (DBH) graphically, rather than an outline of the down log as in the previous subclass.

2.1.3 The "StemContainer" Class

This class definitions provide a so-called *container* class structure that can be used to store a collection or population of "Stem" subclass objects for use in sampling surface simulation. The base class is virtual, but subclasses "downLogs" and "standingTrees" will hold collections of "downLog" and "standingTree" objects, respectively. The "StemContainer" class is something like a list, but knows the types of objects stored within it, and so has the ability to assign subclass-specific functionality based on the subclass type. (See *The "Stem" Class* vignette for more information.)

2.2 The "Tract" Class

We need a geographic representation of an area in two-dimensions on which to build the sampling surface. This can be as simple as a simulated one-hectare plot, or perhaps actually corresponds to some physical geographical location on the ground where measurements have been taken. The "Tract" class implements this through the `raster` and `sp` packages. In fact, the base "Tract" class is a subclass of "RasterLayer", found in the `raster` package. Currently, a "bufferedTract" class exists that simply adds a buffer region to the "Tract" class, and is therefore a subclass of "Tract". Using the "bufferedTract" class for example, we can draw a collection of "Stem" objects from within the buffer area in such a way that the inclusion zones are fully contained within the tract. In the future, other subclasses that automatically handle some of the common boundary overlap methods could be added as well; e.g., "mirageTract" or "walkthroughTract" subclasses could easily be envisoned. (See *The "Tract" Class* vignette for more information.)

2.3 The "ArealSampling" Class

This class allows for the definition of areal sampling methods. In general, each subclass should encapsulate at least the minimal requirements for the sampling method needed to be able to calculate an object's inclusion zone when associated with a "stem" object. The base class is again virtual.Subclasses are defined for each different sampling method and are not currently separated into methods that only apply to down logs or standing trees, because this is taken care of by the "InclusionZone" class constructors. (See *The "ArealSampling" Class* vignette for more information.)

2.4 The "InclusionZone" Class

As already mentioned, the combination of a "Stem" object and an "ArealSampling" class object yields the ability to determine the inclusion zone for the combined objects. The "InclusionZone" class does this through its subclass definitions and object constructors because the base class is again virtual. Currently, there are subclasses for several sampling methods used on "downLog" and "standingTree" objects. For example, the "sausageIZ" class combines the sausage sampling

protocol (Gove and Van Deusen, 2011) with a "downLog" object, while the "horizontalPointIZ" class combines horizontal point sampling and a "standingTree" object. All attributes for the given combination are stored in the object, as well as a "SpatialPolygons" (`sp` package) representation of the perimeter of the inclusion zone itself. (See *The "InclusionZone" Class* vignette for more information.)

2.4.1 The "izContainer" class

As with the "Stem" class, we also need a way to store multiple inclusion zones for a collection of stems in a population. The virtual base class, "izContainer", has two functional subclasses. The "downLogIZs" subclass will store any of the "InclusionZone" subclass objects that pertain to down logs in a container object. Similarly, the "standingTreeIZ" subclass stores a collection of inclusion zones for standing trees. The restriction in both of the "izContainer" subclasses is that all inclusion zones must be the same class of object, so one could not mix inclusion zones generated from the sausage method with any other method, for example. (See *The "InclusionZone" Class* vignette for more information.)

2.5 The "InclusionZoneGrid" Class

If we have objects corresponding to a "Tract" and also "InclusionZone" class of one form or another (which presupposes that we also have an "ArealSampling" object and one or more associated "downLog" or "standingTree" objects), then we are at the stage were we want to combine the inclusion zone and the tract object so that we can subsequently build the sampling surface. The "InclusionZoneGrid" class allows us to do this. It basically takes the background grid attributes from a "Tract" class object and creates a minimal bounding grid large enough to encompass the inclusion zone of an "InclusionZone" object, in the correct spatial juxtaposition. Then, forming the intersection of the inclusion zone polygon with the minimal bounding grid, it assigns the per unit area attributes to all grid cells within the inclusion zone. For a collection of logs, this is done for each individual log. From here, it is a small step to then re-aligning these sub grids to the overall master "Tract" grid, and accumulating the sampling surface. (See *The "InclusionZoneGrid" Class* vignette for more information.)

2.6 The "sampSurf" Class

Of course all of the previous classes now form the chain of steps that leads to simple accumulation of the sampling surface, which is stored in a "sampSurf" class object. The constructors for the "sampSurf" objects will hide the details to varying degrees. For example, you can construct a sampling surface for a given sampling method by using the constructor that specifies the number of logs and a "Tract" object, essentially hiding everything. Other constructors allow different levels of

information to be used in the process. (See *The "sampSurf" Class* vignette for more information.)

3 Coordinate Reference Systems

One point that was not mentioned anywhere above is that all of the spatial objects have slots for the definition of the coordinate reference system (CRS) used to take the measurements. If the `rgdal` package is available, one can take advantage of these to a greater extent. For now, the package just does validity checks to make sure the units used in the stem measurements are the same as those in the projection as best it can without resorting to requiring `rgdal` to function. Right now that amounts to making sure the spatial data are not in geographic form, and that all measurements are either in English or metric—but consistent through all objects that are being worked with at any given time. Both the `raster` and `sp` packages support the use of CRS through class slots and also support `rgdal`, so as this package progresses, the base capability is there, and can be utilized more fully in the future.

4 A Simple Example

Here we use a very simple example showing the steps to creating a sampling surface. It must be stressed that there is a simpler way to generate the surface, as described above, using one of the other constructors, but the following will help give a feel for how the above classes work together to generate the final product. We will use a small example, more detailed examples are found in the vignettes. The results of the following code are found in Figure 1.

```
R> require(sampSurf)
R> tra = Tract(c(x=25, y=25), cellSize = 0.5, units = 'English',
+              description = 'a small plot')
R> (btr = bufferedTract(bufferWidth=5, tract=tra))
```

```
------------------------------------------------------------
a small plot
------------------------------------------------------------
Measurement units = English
Area in square feet = 625 (0.014348026 acres)

class       : bufferedTract
dimensions  : 50, 50, 2500  (nrow, ncol, ncell)
resolution  : 0.5, 0.5  (x, y)
extent      : 0, 25, 0, 25  (xmin, xmax, ymin, ymax)
```

```
coord. ref. : NA
values      : in memory
min value   : 0
max value   : 0

Buffer width =  5
```

```
R> dlogs = downLogs(5, btr, units = 'English',
+                   buttDiams = c(4,10), logLens = c(2,10))
R> listSUIZ = lapply(dlogs@logs, 'standUpIZ', plotRadius = 2.5)
R> sapply(listSUIZ, class)
```

```
      log.1       log.2       log.3       log.4       log.5
"standUpIZ" "standUpIZ" "standUpIZ" "standUpIZ" "standUpIZ"
```

```
R> izsSU = downLogIZs(listSUIZ)
R> ssSU = sampSurf(izsSU, btr)
```

```
Number of logs in collection = 5
Heaping log: 1,2,3,4,5,
```

```
R> summary(ssSU)
```

```
Object of class: sampSurf
------------------------------------------------------------
sampling surface object
------------------------------------------------------------

Inclusion zone objects: standUpIZ
Measurement units = English
Number of logs = 5
True log volume = 9.0622734 cubic feet
True log length = 30.7 feet
True log surface area = 56.372908 square feet
True log coverage area = 17.911536 square feet
True log biomass = NA
True log carbon = NA

Estimate attribute: volume
```

```
Surface statistics...
  mean = 9.1094315
  bias = 0.047158106
  bias percent = 0.52037832
  sum = 22773.579
  var = 585.67275
  st. dev. = 24.200677
  cv % = 265.66616
  surface max = 117.35777
  total # grid cells = 2500
  grid cell resolution (x & y) = 0.5 feet
  # of background cells (zero) = 2111
  # of inclusion zone cells = 389
```

The above hides the creation of an "ArealSampling" class object—in this case a fixed-area circular plot—inside the creation of the "standUpIZ" object construction, but it is indeed there, as witnessed by the required `plotRadius` argument. The following gives a short explanation of the steps in the above example.

1. The first line of code just makes sure the `sampSurf` package has been loaded.
2. In the next two lines of code, an object of class "Tract" is first created. Its dimensions are 25×25 feet, with a half-foot resolution. The minimum extent of the bounding box for the tract will be at $(0, 0)$, and by default the values for all cells are set to zero. The second line creates a "bufferedTract" object from the "Tract" object, with a buffer width of five feet.
3. The following line makes a collection of "downLog" objects and stores them in a "downLogs" container class object. We make a collection of short logs to allow everything to fit nicely into this small plot (tract), and make sure the large-end diameters in inches are something reasonable. In each case, dimensions are sampled from the limits provided.
4. The next step is to make a collection of "InclusionZone" objects from the collection of "downLog" objects using a given "ArealSampling" method—in this case using the so-called stand-up protocol for circular plots. To do this simply, we use the **R** `lapply` command, which takes each of the individual "downLog" objects that are stored within a list slot in the "downLogs" container, and applies the `standUpIZ` method to them. The result is confirmed in the following step: each of the resulting objects in the list is of class "standUpIZ".
5. The penultimate step is to turn the list created in the previous step into a "downLogIZs" container object. Note that the current and previous steps could have easily been combined into one **R** command as: `izsSU = downLogIZs(lapply(dlogs@logs, 'standUpIZ', plotRadius=2.5))`, but we have separated them here for clarity in order to clearly show the outcome of the `lapply` command.

6. Finally we simply create the "sampSurf" object from the "downLogIZs" collection and the "bufferedTract" object. And show how the `summary` generic has been adapted to print some statistics on the sampling surface (note that the stand-up method is unbiased).

The above example is the long way to create a sampling surface, but allows the most control and demonstrates the steps discussed above with respect to the individual classes and constructors in the package. As noted above and in the "sampSurf" class vignette, this can all be done with one call to an alternative `sampSurf` constructor function by specifying the desired number of logs, their attributes, and a "Tract" object.

Finally, the sampling surface object can be plotted simply as...

```
R> plot(ssSU, useImage=FALSE)
```

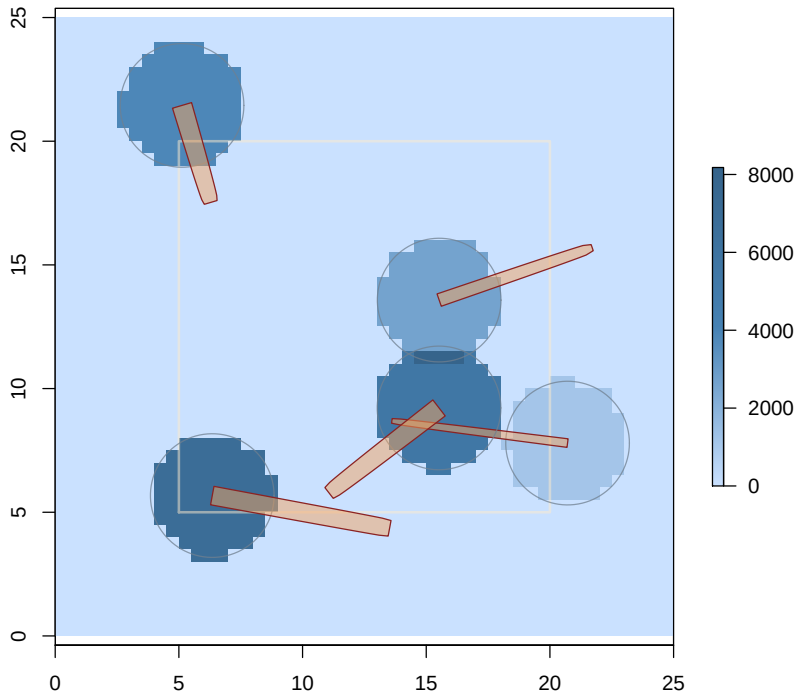

Figure 1: Simple generic "sampSurf" object with some random logs.

5 Summary

The `sampSurf` package currently has a number of different areal sampling methods available for down logs. The design of the package through the class structure and the underlying flexibility of S4, should allow other methods to be added without an inordinate amount of programming, because the basic foundation is there. More information on each class and associated constructors are found in the vignettes. The best source for complete documentation of classes, constructors and other generic functions is in the package help files.

6 Appendix: A Brief Introduction to S4

In the above introduction, we have been speaking of terms that may not be all that familiar to the casual R user. Every object in R has a class, just like it has a type or mode (for example, try `class(get('+'))`, then substitute `typeof` or `mode` for `class`). But since S4 is a true object-oriented system, the class structure of objects under S4 can take on a new dimension through inheritance, which is not found in the more traditional R paradigm (known as S3). Inheritance allows us to define a general base class and refine it by making subclasses of the base class that share all of its attributes, plus new ones. For example, maybe someone has defined a class called "Tree" in S4, but did not make allowance for more information that you now need, say, degree of lean, for example. One can simply define a "leaningTree" subclass of "Tree" that inherits all of the attributes of this base class, but adds a slot for degree of lean. One can think of the inheritance hierarchy as a tree with different nodes or leaves forming the classes. In the example, the "Tree" class is a *superclass* of "leaningTree"; or, if you like, "leaningTree" is a *subclass* of "Tree". The subclass is always an object of its superclass because is shares all the slots of the superclass, but the superclass object is not an object of the subclass, because more slots, and therefore functionality have been added in the subclass that an object of class "Tree" would not know how to handle.

Functionality for classes is given through generic functions that have methods defined for the desired classes. The link between the two—or how the method knows to automatically act on a given class—is derived via the method's *signature*. The signature is composed of one or more arguments to the function that are used in method dispatch: the act of determining the correct method to apply to the signature arguments based on their classes (all taken care of behind the scenes by R). To continue our little example, suppose we want a method to automatically print certain components of a "Tree" object when one types the object's name at the command line and hits the enter key, because the object is quite large and there is no need to print everything. We would define a method for the generic `show` function that has a signature argument of class "Tree" encapsulating exactly what is to be printed when showing the object. Furthermore, because of inheritance, one can also define a method for "leaningTree" objects which will use the existing functionality of the `show` method defined for the "Tree" class, plus whatever other slots were added in the subclass. So subclasses are extensions of superclasses that simply provide more functionality in terms of object slots and methods.

One last definition concerns the term *slot*, which we have used above. A slot is a named component of a class where something is stored. It is somewhat analogous to the components in a `list` object. In a `list` object we can access a component with the `$` operator. In S4 objects, slots are accessed using the `@` operator, or the `slot` function. But this is a trivial difference. Slots in S4 objects are strongly typed, allowing only objects of a given class (or union of classes), which is determined on class creation by the class designer, to be associated with the value for each slot. Therefore, the S4 system can guarantee that the information contained within a slot is in the expected form for a valid object. Object validity checking is yet another aspect and will not be covered here. There is far more to know about S4, in fact it is quite a fully functional object-oriented paradigm within R. Much more information is provided in documents on the web, and in Chambers (2008). The newly added "Reference" classes, that are evidently built on S4 present another powerful addition to the object oriented toolkit within R.

References

J. M. Chambers. *Software for Data analysis: Programing with R*. Springer, 2008. 11

M. J. Ducey, M. S. Williams, J. H. Gove, and H. T. Valentine. Simultaneous unbiased estimates of multiple downed wood attributes in perpendicular distance sampling. *Canadian Journal of Forest Research*, 38:2044–2051, 2008. 2

J. H. Gove and P. C. Van Deusen. On fixed-area plot sampling for downed coarse woody debris. *Forestry*, 84(2):109–117, 2011. 1, 3, 5

J. H. Gove, A. Ringvall, G. Ståhl, and M. J. Ducey. Point relascope sampling of downed coarse woody debris. *Canadian Journal of Forest Research*, 29(11):1718–1726, 1999. 2

J. H. Gove, M. S. Williams, G. Ståhl, and M. J. Ducey. Critical point relascope sampling for unbiased volume estimation of downed coarse woody debris. *Forestry*, 78:417–431, 2005. 1

J. H. Gove, M. J. Ducey, and H. T. Valentine. A distance limited method for sampling downed coarse woody debris. *Journal of Applied Ecology*, 2012. (In preparation). 1, 2

H. T. Valentine, J. H. Gove, and T. G. Gregoire. Monte Carlo approaches to sampling forested tracts with lines or points. *Canadian Journal of Forest Research*, 31:1410–1424, 2001. 1

M. S. Williams. New approach to areal sampling in ecological surveys. *Forest Ecology and Management*, 154:11–22, 2001a. 1

M. S. Williams. Nonuniform random sampling: an alternative method of variance reduction for forest surveys. *Canadian Journal of Forest Research*, 31:2080–2088, 2001b. 1

M. S. Williams and J. H. Gove. Perpendicular distance sampling: an alternative method for sampling downed coarse woody debris. *Canadian Journal of Forest Research*, 33:1564–1579, 2003. 1, 2

M. S. Williams, M. J. Ducey, and J. H. Gove. Assessing surface area of coarse woody debris with line intersect and perpendicular distance sampling. *Canadian Journal of Forest Research*, 35: 949–960, 2005. 2