# Extending the sampSurf Package

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

Wednesday 28th December, 2011
4:55pm

## Contents

**1 Introduction** 1
**2 The Major Components of sampSurf** 2
2.1 General Steps in Package Extension . . . 2
2.1.1 "ArealSampling" subclass . . . . . 3
2.1.2 "ArealSampling" constructor . . . 4
2.1.3 "InclusionZone" subclass . . . . . 5
2.1.4 "downLogIZs" container class constructor . . . . . 6
2.1.5 "InclusionZoneGrid" constructor . 7
2.1.6 "sampSurf" constructor . . . . . 8
**3 Other Generics Requiring Methods** 8
3.1 The plot and perimeter methods . . . . 9
3.2 The summary and show methods . . . . . 9
**4 R Documentation** 10
4.1 *PDF* Files . . . . . 10
4.2 *Rd* Files . . . . . 10
4.2.1 New *Rd* files . . . . . 11
4.2.2 Additions to existing *Rd* files . . . 11
**5 Other** 11
**6 Adding A New Estimate Variable** 12

## 1 Introduction

By now, if you are at this point, it is pretty evident that **sampSurf** was designed (using the term loosely) with a class-constructor structure that should allow it to be extended fairly (again loosely applied) easily to incorporate other sampling methods. In what follows we will look at how one might add the PRS sampling method for down woody debris to the package. Only an outline of the procedure will be given here. For the details, please look at the code in the files mentioned below for PRS and PDS (or other method) to get a "model" of what to do.[1]

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

[1]Please note that this vignette is only for people who want to add a method to the package (please contact the author if so) or extend the package in your own working environment. In the latter case, the programming components

If you've gotten this far, you undoubtedly have read in the vignettes or the code where the package is basically all written using the S4 paradigm. We will not belabor this, and the assumption will be made that if you are looking to extend `sampSurf` by adding a new sampling method, you understand the basics of S4 classes and methods. The package code is a good place to get more information as mentioned above. The S4 paradigm was chosen for a reason: it is a truly flexible and extensible object-oriented way to program in R. For example, the use of inheritance in both classes and the extension of methods (i.e., `callNextmethod`) for generic functions is very powerful and will save much in the way of programming time when done correctly. We give a little example of this in the S4 section of *The sampSurf Package Overview*. I have read where S4 is considered slow, or was when it first came out. This really is not a valid argument when one considers the huge benefits it brings for package integrity (e.g., validity checking) and source code reuse; our computers get faster all the time, and the small amount of extra time S4 might take is minimal compared to the benefits it brings to coding in R.

If you add an extension to `sampSurf` that ends up being something that could be valuable to other users, it is possible to contribute the code to the *R-Forge* site, and this would be most appreciated.

If you follow the steps below, and follow along with existing code, then your extension should integrate seamlessly with the rest of the package. For example, the container classes should "just work" as long as you have maintained the correct class inheritance structure.

# 2 The Major Components of sampSurf

The major components of `sampSurf` are discussed in the *The sampSurf Package Overview* vignette. Please refer to that document for a quick refresher if needed. There are several ways one could extend the package. First, one could add new subclasses to the "Stem" class for further refinement of either down logs or standing trees—or perhaps add a whole new category like woody shrubs. Another possibility would be to add a new subclass to the "Tract" class, perhaps for a different definition of a buffered tract; options here might be subclasses that automatically know how to do mirage or walkthrough boundary overlap correction. These are certainly possible, but are not discussed here at present. Please see the code for examples on how this might be done. In the following we restrict our attention to adding new sampling methods, as this is the most common extension that will be made to the package.

## 2.1 General Steps in Package Extension

Below are the general steps that are required in adding a new sampling method to `sampSurf`.

mentioned in the following will be necessary for the most part (but not all), and the documentation components will be unnecessary.

1. Add the class definition as a subclass of "ArealSampling".
2. Add a constructor method to construct objects of the class rather than having the user rely on `new`, which is too generic and error-prone.
3. Add the class definition for the inclusion zone as a subclass of "InclusionZone". If we are dealing with down logs, for example, this will actually be a subclass of "downLogIZ".
4. Add the `InclusionZoneGrid` method; note that there is nothing to modify in the class definition for inclusion areas that are constant height, as in PRS. In fact, most variable height sampling methods like omnibus PDS and DLMCS will require no class modification either. The "csFullInclusionZoneGrid" is the exception because the individual chainsaw "InclusionZoneGrid" objects get stored within a suasage inclusion zone for reference, and it becomes something resembling a container object as a result.
5. Add the name of the "InclusionZone" object to the `sampSurf` constructor that has a numeric signature as the first argument.

### 2.1.1 "ArealSampling" subclass

Each new sampling method needs its own class definition. The "ArealSampling" class was setup to be the source of these definitions. It is a virtual class which forms the basis for all of the sampling methods that are available in `sampSurf`. To extend this one must define a new subclass with the `contains = 'ArealSampling'` argument in the class definition given to `setClass`. That's the main constraint; after this, it is up to you to determine what makes sense to include in the class definition for a given method. For example, in PRS, we have the relascope angle and the squared-length factor among other slots. Generally, anything you include here should be easily calculated within the constructor without too much extra input.

```
R> showClass('pointRelascope')
```

```
Class "pointRelascope" [package "sampSurf"]

Slots:

Name:  angleDegrees angleRadians          phi     slFactor     rwFactor
Class:      numeric      numeric      numeric      numeric      numeric

Name:   description        units
Class:    character    character

Extends: "ArealSampling"
```

A very important point to keep in mind is that the "ArealSampling" class is not the place to have any information about a method that relies on knowing anything about the individuals to be sampled. For instance, in the case of PRS, we would not put any slots in the class definition that rely on knowing the log length. This is why the "ArealSampling" and "InclusionZone" classes were separated. The inclusion zone *does* depend on both the sampling method and the measurements on the individual log or tree in question, so the "InclusionZone" subclass is where one would put, e.g., the area of the inclusion zone and its perimeter under PRS and PDS.

Finally, here and in the following, it is extremely important to include as many validity checks in the class definition as possible. This can not be overstated. If the object created by the constructor is bad to begin with, then there is no telling what will happen afterwards. The `validity = function(object)` section of the `setClass` call is the place to do as much of this as possible because people still might try to use `new` to construct and object, and problems will be caught here if they do. Some checking might also have to be done in the object constructor to make sure arguments passed are valid before use, and may even be redundent, but this is fine, there can't really be too much of this.

### 2.1.2 "ArealSampling" constructor

As mentioned above, it really is helpful to the end user to have a constructor—preferably of the same name as the class for the resultant object—rather than rely on the use of `new`. The latter is frought with possible misapplication, especially for complex objects. The constructor should operate on the minimal necessary arguments, offering defaults where possible, to construct a valid object. As mentioned above, some pre-checking of the validity of arguments passed may be necessary and is encouraged.

The following example shows how the PRS constructor can be used to construct a list of multiple objects at one time, based on the reach:width ratios for the angles in question[2]...

```
R> lprs = lapply(sapply(1:4, function(nu).StemEnv$rad2Deg(2*atan(1/nu))),
+                function(x) pointRelascope(x, units='English'))
R> sapply(lprs, class)
```

```
[1] "pointRelascope" "pointRelascope" "pointRelascope" "pointRelascope"
```

```
R> t(sapply(lprs, function(x) c(x@rwFactor, x@angleDegrees, x@slFactor)))
```

```
     [,1]      [,2]       [,3]
[1,]    1 90.000000 55462.3146
```

[2] See `?.StemEnv`; the function `rad2Deg` that resides there simply converts from radians to degrees.

```
[2,]    2 53.130102 20694.3744
[3,]    3 36.869898 10531.3082
[4,]    4 28.072487  6290.8019
```

The constructor is quite basic. It is advisable to make each constructor a generic function and add a method to construct the object. In this way, others could add subclasses or other methods to simply extend the constructor, making life easier. In addition, someone might have a different way to construct the object which could be coded based on a change in signature, and the opportunity to do so should be available.

```
R> isGeneric('pointRelascope')
```

```
[1] TRUE
```

```
R> showMethods('pointRelascope')
```

```
Function: pointRelascope (package sampSurf)
angleDegrees="numeric"
```

In the above example, we used the signature argument `angleDegrees`, as there is probably little likelihood that anyone will want to extend this constructor using a different type of signature argument. In general, if this is not the case and multiple constructors (see the "downLogs" constructors) are more likely, then it might be wise to make the signature argument(s) more generic in its name, such as `object`.

### 2.1.3 "InclusionZone" subclass

This is where we develop the class configuration necessary to have an object that acts like an inclusion zone. These objects all are very similar regardless of whether the zone will eventually contain a constant or variable surface. We want a class definition and associated constructor that defines the zone geometrically so it can be subsequently overlayed onto the "Tract" grid for the assignment of the individual grid cells within the zone. Therefore, in most cases, the object and constructor will be very similar to what we have defined for PRS below.

The newly generated class for PRS looks like the following...

```
R> showClass('pointRelascopeIZ')
```

```
Class "pointRelascopeIZ" [package "sampSurf"]

Slots:

Name:              prs      dualCircle          radius            area
Class:  pointRelascope          matrix         numeric         numeric

Name:        perimeter      pgDualArea     dualCenters         downLog
Class: SpatialPolygons         numeric          matrix         downLog

Name:      description           units            bbox         spUnits
Class:       character       character          matrix             CRS

Name:        puaBlowup    puaEstimates       userExtra
Class:         numeric            list             ANY

Extends:
Class "downLogIZ", directly
Class "InclusionZone", by class "downLogIZ", distance 2
```

As usual, a constuctor must be written for this class, preferably with the same name. For PRS we have the following as an example...

```
R> etract = Tract(c(x=100,y=100), cellSize=0.5, units='English')
R> ebuffTr = bufferedTract(15, etract)
R> dlgs = downLogs(10, ebuffTr, units='English', buttDiam=c(10,20), logLens=c(5,10))
R> iz.prs = pointRelascopeIZ(dlgs@logs$log.1, lprs[[1]])
```

In the above, and in some of the other earlier class definitions, I have used slot names specific to the sampling method. For example, the `dualCircle` slot contains the matrix representation of the inclusion zone perimeter. When writing the PRS code, however, I decided it would be much better to try to make some of these shared slot names more standard. So for example, the above slot now becomes `izPerim` in a PRS object. In addition, the names for polygons within the `perimeter` slot have been standardized, rather than reflecting the sampling method. Please see the code for examples. The older classes (prior to PDS) will remain as they are, but future classes should follow this more consistent approach if possible.

### 2.1.4 "downLogIZs" container class constructor

There is nothing that requires changing in this class or its constructor, it should be able to handle all existing and new "InclusionZone" class objects as long as there is a `perimeter` function available

for the class.

```
R> lprs.izs = lapply(dlgs@logs, function(x) pointRelascopeIZ(x, prs=lprs[[1]]))
R> dl.izs = downLogIZs(lprs.izs)
R> dl.izs
```

```
Container object of class: downLogIZs
------------------------------------------------------------

------------------------------------------------------------
There are 10 inclusion zones in the population
Inclusion zones are of class: pointRelascopeIZ
Units of measurement:  English
Summary of inclusion zone areas in square feet...
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.        var
 22.145000  24.542000  36.265000  38.506000  51.976000  57.818000 224.699721
      SDev
 14.989987

Encapulating bounding box...
        min       max
x 24.860733 87.573264
y 15.348698 75.056959
```

### 2.1.5 "InclusionZoneGrid" constructor

This is where the actual per unit area estimates get assigned to each grid cell within the inclusion zone. The base constructors for this class are methods of the `izGrid` generic.[3] There are two possible situations that might arise...

1. *Constant* sampling surface within each object's inclusion zone: In this case simply copy one of the methods that is similar here and make your changes, any method that generates constant inclusion zone height will do, such as the sausage method, and modify the signature for the new class: in this case "pointRelascopeIZ".

2. *Variable* sampling surface within each object's inclusion zone: This case is a bit more involved. Each method will have a different way of assigning the per unit area estimates to the individual grid cells. Examples are shown for a number of methods such as omnibus PDS and distance limited PDS (DLPDS) or distance limited Monte Carlo (DLMCS). The extreme is the full

[3]The exception being `izGridCSFull` for the full chainsaw method.

chainsaw method, but the way this was set up was quite different from the other variable-height methods because I elected to keep track of the intersection objects in this method, and because the inclusion zone is actually a "SausageIZ" object, so it is perhaps not a good model to follow in general. Sampling methods that generate variable height surfaces are more work to code for sure, but in all cases so far encountered, the information reqired to generate estimates for each grid cell is readily extractable. For example, with regard to functions of perpendicular log diameters (variants of PDS and DLMCS), the diameters are all available through the taper function (either in default or spline form). The code presents examples of back-transforming logs to the "canonical" position, to make it simpler to get at certain attributes like perpendicular diameters. There is undoubtedly a simpler way to do this, but these examples can be used as guides.

In the end, we should arrive at a constructor function that can be called as, e.g.,...

```
R> prsIZG = izGrid(iz.prs, ebuffTr)
```

### 2.1.6 "sampSurf" constructor

At present, there are two constructors for class "sampSurf" and neither requires any modification (unless you come up with something resembling the full chainsaw method). Remember, you must pass along other required arguments to underlying constructors as shown in the following, where both constructors are demonstrated with the new PRS addition...

```
R> prs.ss = sampSurf(dliz, ebuffTr)
R> prs.ss2 = sampSurf(4, ebuffTr, iZone='pointRelascopeIZ',
+                     units='English', prs=lprs[[1]])
```

Note in the last constructor that we must pass along the "ArealSampling" point relascope object so that the `pointRelascopeIZ` constructor has this information to work with. We also must pass along the units, as 'metric' is the default in generating sample trees with `downLogs`.

## 3 Other Generics Requiring Methods

There are other generics like plot, summary, and perimeter that also has to be updated for classes where extensions have been made.

## 3.1 The `plot` and `perimeter` methods

In order to visualize new extentions correctly, the `plot` generic must have a method for each class of object that we want to display graphically. These include the following, related to the special classes in `sampSurf`...

1. "ArealSampling": The only time this would require a new plot method for an extension is in th case of something like the circular plot (say a rectangular plot), where the size of the plot does not depend on some attribute of the downed log. This would be true for both `plot` and `perimeter`.
2. "InclusionZone": The methods for protocols other than, e.g., chainsaw, are all very similar. Just duplicate from say, `sausageIZ`, and make any additions. This is what was done for `pointRelascopeIZ`. This would be true for both `plot` and `perimeter`. Please note above that for the "downLogIZs" container to work, there must already be an associated `perimeter` method for the "InclusionZone" objects to be stored.

Plot methods for "InclusionZoneGrid" and "sampSurf" classes should work without modification, providing one has correctly filled each of the class slots as expected.

## 3.2 The `summary` and `show` methods

The `summary` and `show` generics may require methods for the new classes; for example...

1. "ArealSampling": One should extend the base class functionality to print relevant information from the class object. Regard the methods for "pointRelascope" or any other for an example of how this was extened. As a default, the `show` method simply calls the `summary` method, so it does not require any additions or changes.
2. "InclusionZone": Again, we want to extend the base method by adding information that is relevant to the class. See the code for examples. Again, as a default, the `show` method simply calls the `summary` method, so it does not require any additions or changes.

Finally, in case it is not obvious, when you `print` and object at the command line, `print` simply calls the `show` method by default, so everything will work correctly here.

# 4 R Documentation

If a new method makes it into the `sampSurf` package, then there will be documentation changes that are required. If one is simply extending the methods above for one's own use, this step is not required.

The documentation comes in two flavors: *PDF* vignettes and *Rd* help files. We go over each of these below...

## 4.1 PDF Files

It is probably a good idea to put examples of any new methods into the vignettes. Undoubtedly, the new methods will have their own arguments that differ from extant methods that would require a little example. And in general, having an example available is just a good idea so people can use it to go by when writing their own code. The following *PDF* vignettes should be modified for new sampling methods...

1. *The "Areal Sampling" Class* should have an example of the new subclass. For example see the "pointRelascope" class addition to the original document. The documentation should include an explanation of the class slots and an example of creating an object at a minimum.
2. *The "InclusionZone" Class* should also have examples showing the class slots, how to create an object from the class, and then how to plot this object. Again, see the "pointRelascopeIZ" class addition to the original document.
3. *The "InclusionZoneGrid" Class* should have a simple example for the new class of "InclusionZone" object that has been added. There are plenty of examples in the documentation to go by for this.
4. *The "sampSurf" Class* additions are optional. In the case of PRS, an example was added to show what needs to be passed along to one of the constructors as detailed above just so people would understand the extra requirements.

The other vignettes do not require additions. One might add something for a new technique to these, but that is not necessary.

## 4.2 Rd Files

This is the laborious part. For each of the changes made in the "Major Components Of sampSurf" and "Other Generics..." sections above, we will need new or augmented *Rd* documentation. There

is no way around this, it just has to be done for a complete package.

### 4.2.1 New Rd files

Several new *Rd* files will be required...

1. The new "ArealSampling" class definition, the new generic that creates the class, and the new method for object construction. For example, for PRS the *Rd* files that require creation are: `pointRelascope-class`, `pointRelascope` and `pointRelascope-methods`, for the class definition, generic, and constructor method, respectively.
2. The new "InclusionZone" class definition, the new generic that creates the class, and the new method for object construction. For example, for PRS the *Rd* files that require creation are: `pointRelascopeIZ-class`, `pointRelascopeIZ` and `pointRelascopeIZ-methods`, for the class definition, generic, and constructor method, respectively.

### 4.2.2 Additions to existing Rd files

Additions to existing *Rd* files will be required for...

1. The `ArealSampling-class` file should be modified to link to the new method.
2. The `sampSurf-package` file should be modified to contain links to all of the documentation files added above in the same model as what is already in that file for other methods.
3. The `downLogIZs`, `downLogIZs-class` and `InclusionZone-class` files require mention of the new sampling method as a link in the "See Also" section.
4. The `perimeter-methods` and `plot-methods` files require information on the new sampling method.
5. The `izGrid-methods` file must have the appropriate addition.

## 5 Other

Again, if the new method is to be added to the package, then we must make the appropriate changes to the *DESCRIPTION* and *NAMESPACE* files. If code has been added to existing files, then nothing needs to be done to the former. However, all classes and methods must be explicitly `export`ed from the package namespace in the *NAMESPACE* file, or they will be unavailable to users.

# 6 Adding A New Estimate Variable

In this case, we want to add a new attribute or quantity to be estimated like volume or carbon. This is most easily done in the package environment, rather than in an individual's own workspace. Hopefully what is provided will meet the needs of most users and future additions will not be required. The following steps should be used to add the new variable (below only specifies changes for down logs, similar adjustments will be necessary for standing trees)...

1. Add the new variable to the `puaEstimates` vector in the `.StemEnv` environment.
2. The appropriate "Stem" subclass definition should be modified (e.g., "downLog") to have a slot for the new information. Please also modify the associated validity and initialize routines as required.
3. The constructor(s) for the above object needs to handle the new input. This will probably, but not always, require a new argument and some code.
4. The `summary` method for the above class, and associated container class should be modified.
5. The `.statsDownLogs` function will require addition of the new variable for the "downLogs" container class.
6. All "InclusionZone" constructor methods will need to add the new variable—not a trivial undertaking.
7. The associated inclusion zone `summary` method will need modification for the "downLogIZ" class and for the chainsaw method.
8. The "InclusionZoneGrid" constructor for methods that have non-constant surface will require modification. These include omnibus PDS (see its `izGrid` method) and the chainsaw method (see the `chainsawSliver` routine).
9. The main "sampSurf" constructor will require addition of the new "truth" value for this variable estimate.
10. The "sampSurf" `summary` method will also require the "True" value in `cat` statements and in the `truth = switch` statement.