Monte Carlo Sampling Methods in sampSurf*

J. H. Gove

USDA Forest Service, Northern Research Station, 271 Mast Road, Durham, NH 03824 USA

(603) 868-7667; e-mail: jgove@fs.fed.us

Friday 18th October, 2013 3:23pm

Contents

*R sampSurf package vignette series paper: http://sampsurf.r-forge.r-project.org/.

1 Introduction

Basic Monte Carlo sampling methods, which include crude Monte Carlo (CMC), importance sampling (IS) and control variate sampling (CV) are available in `sampSurf` for estimating volume of individual "downLog" or "standingTree" objects. In addition, antithetic sampling (AS) is also available for each of the above variants. There are a number of papers explaining how to use Monte Carlo sampling methods to estimate volume within a log or tree bole. A good, accessible overview with examples can be found in Gregoire and Valentine (2008, p. 106). The implementation of the methods in `sampSurf` follows these authors closely.

Essentially, the Monte Carlo method as applied to standing tree boles or down logs consists of estimating an integral quantity. Here, the integral is segment volume; *viz.*,

$$v = \int_a^b \rho(h)\mathrm{d}h \tag{1}$$

where $\rho(h)$ is cross-sectional area at height h (or length l). When volume for the entire bole is to be estimated, for example, then $a = 0$ and $b = H$ (or L), the total stem height (or length). Recall that in `sampSurf`, all diameters are stored in the same units as length/height and the integral above reflects that correspondence.

In the following, the basic class structure (Figure 1) and methods for creating objects is presented, along with other routines for plotting and summarization.

The exposition in the first several sections (§3–11) is based on sampling within a "Stem" subclass object (Gove, 2011a) to estimate segment volume, independently of any areal sampling method. However, the Monte Carlo methods described here can also be used in connection with areal methods, as a way to unbiasedly estimate volume in a second stage at each sampling point for any stems whose inclusion zones overlap the sample point (i.e., grid cell centers). Examples of the latter will be given in §12.

2 Notation

The notation follows loosely from Gregoire and Valentine (2008). For example,

n_s = the number of Monte Carlo samples in the bole segment: $s = 1, \ldots, n_s$.

g_s = proxy cross-sectional area at the sth sample height, h_s.

G = proxy bole (segment) volume estimate: $G = \int_a^b g_s(x)\mathrm{d}x$, where a and b are either the base and top of the tree, or define the bottom and top heights of some segment along the bole, and are stored in the `segBnds` slot of the object.

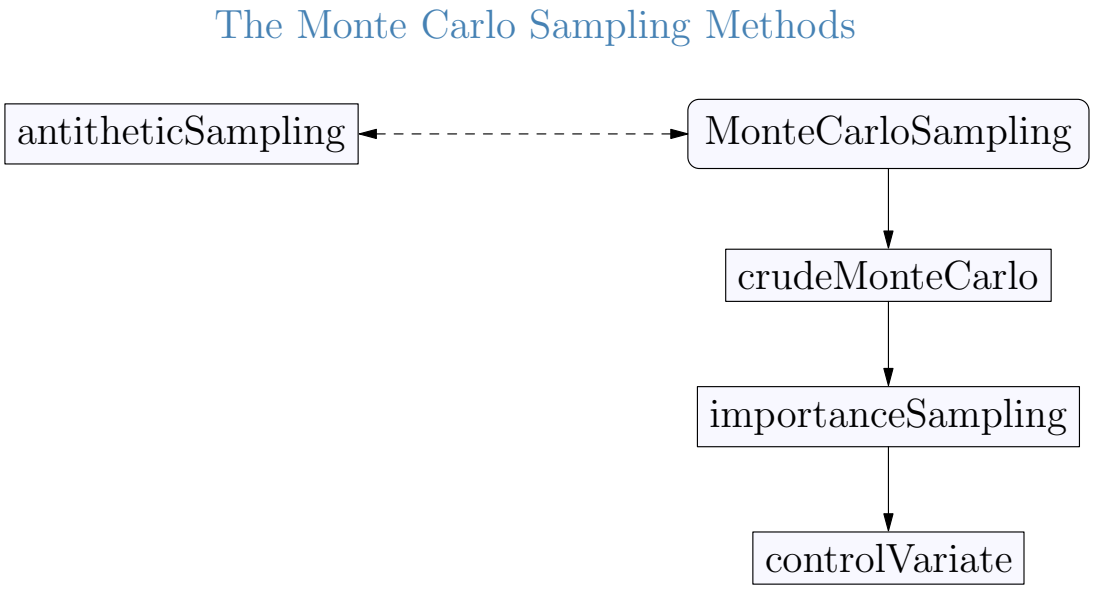

Figure 1: The class structure for Monte Carlo sampling methods for bole volume in `SampSurf`. Note that the connection between antithetic sampling and the "MonteCarloSampling" subclasses is shown (dashed; see §7)

In class descriptions below, several of the slot names use an "`.s`" to denote a vector quantity that indexes to the sth sample point on the bole; n_s above is the exception to this rule and is scalar in the objects as already described.

3 The "MonteCarloSampling" Class

The "MonteCarloSampling" class is a virtual base class for the three sampling methods shown in Figure 1; antithetic sampling works with any of these three methods and is therefore a separate class. The base slots for the structure are...

```
R> showClass('MonteCarloSampling')
```

```
Virtual Class "MonteCarloSampling" [package "sampSurf"]

Slots:
```

```
Name:         stem     segBnds         n.s   startSeed         u.s description
Class:        Stem     numeric     numeric     numeric     numeric   character

Name:     userArgs
Class:        list

Known Subclasses:
Class "crudeMonteCarlo", directly
Class "importanceSampling", by class "crudeMonteCarlo", distance 2
Class "controlVariate", by class "importanceSampling", distance 3
```

3.1 Class slots

- *stem*: A "Stem" subclass object; i.e., "downLog" or "standingTree" (see Gove, 2011a).
- *segBnds*: A vector of length two giving the lower and upper height/length bounds for volume estimation within the bole. All of the following slot definitions below are relative to the segment of the bole defined by these bounds. In the definition for G above, these bounds correspond to the limits of integration a and b.
- *n.s*: The number of Monte Carlo samples n_s (scalar).
- *startSeed*: The scalar seed for the random number generator used in the call to the class constructor. Please see the documentation in `initRandomSeed` in `sampSurf` for possible values and their meaning. Suffice it to say that storing this in the object allows for object replication. Note that if `startSeed = NA`, then the seed is not replicable, but the sampling run is by using the random numbers in the `u.s` slot.
- *u.s*: The uniform random numbers (u_s) used in selecting the sampling points along the bole.
- *description*: A `character` description of the object if desired (defaults are given for each class).
- *userArgs*: Some proxy functions have extra arguments (§8) that are required when called from the constructor methods. This slot stores these arguments and their values from the call. This is necessary, e.g., for re-applying a given Monte Carlo method to the $(1-u_s)$ points in antithetic sampling.

4 The "crudeMonteCarlo" Class

Crude Monte Carlo sampling is the simplest method for estimating the integral (1). The volume estimator and its associate variance estimator are given as (Gregoire and Valentine, 2008, p. 99)

$$\hat{v}_s = (b-a)\rho(h_s) \tag{2}$$

$$\hat{\bar{v}} = \frac{b-a}{n_s}\sum_{s=1}^{n_s}\hat{v}_s \tag{3}$$

$$\hat{\text{var}}(\hat{\bar{v}}) = \frac{\sum_{s=1}^{n_s}\left(\hat{v}_s - \hat{\bar{v}}\right)^2}{n_s(n_s-1)} \tag{4}$$

where $h_s = a + (b-a)u_s$ are the random heights chosen for sampling along the bole.

4.1 Class slots

The class slots are (including those from the "MonteCarloSampling" base class)...

```
R> showClass('crudeMonteCarlo')
```

```
Class "crudeMonteCarlo" [package "sampSurf"]

Slots:

Name:        proxy      diam.s       rho.s       hgt.s       vol.s      volEst
Class:   character     numeric     numeric     numeric     numeric     numeric

Name:       volVar       ci.lo       ci.up  alphaLevel     trueVol   relErrPct
Class:     numeric     numeric     numeric     numeric     numeric     numeric

Name:         stem     segBnds         n.s   startSeed         u.s description
Class:        Stem     numeric     numeric     numeric     numeric   character

Name:     userArgs
Class:        list

Extends: "MonteCarloSampling"

Known Subclasses:
Class "importanceSampling", directly
Class "controlVariate", by class "importanceSampling", distance 2
```

where the additional slots added in this class are defined as...

- *proxy*: The proxy function identifier used; more details are given in §8.
- *diam.s*: A vector of diameters (d_s) at the sampled heights `hgt.s`.
- *rho.s*: A vector of cross-sectional areas ($\rho_s \equiv \rho(h_s)$) at the sampled heights.
- *hgt.s*: A vector of sampled heights (h_s).
- *vol.s*: A vector of volume estimates ($\hat{v}_s$) associated with the sampled heights.
- *volEst*: The sample mean volume estimate for the bole segment $\hat{\bar{v}}$.
- *volVar*: The within-bole variance estimate of $\hat{\bar{v}}$ for the segment, $\hat{\text{var}}(\hat{\bar{v}})$ as defined in (4).
- *ci.lo*: The lower $1-\alpha$ confidence interval on the bole volume estimate: $\hat{\bar{v}} - t_{n_s-1}^{1-\alpha/2}\sqrt{\hat{\text{var}}(\hat{\bar{v}})}$.
- *ci.up*: The upper $1-\alpha$ confidence interval on the bole volume estimate: $\hat{\bar{v}} + t_{n_s-1}^{1-\alpha/2}\sqrt{\hat{\text{var}}(\hat{\bar{v}})}$.
- *alphaLevel*: The two-tailed α level used in confidence interval construction.
- *trueVol*: The true volume ($\mathfrak{v}$) for the stem segment being estimated.
- *relErrPct*: The relative error $100 \times (\hat{\bar{v}} - \mathfrak{v})/\mathfrak{v}$ in percent.

Note that the quantities above will vary based on the sampling method, e.g., compare $\hat{\bar{v}}$ for CMC (3) and IS (6), or $\hat{v}_s$ for CV (7). In addition, the "`cmcProxy`" is always used for CMC, but it will differ for the other Monte Carlo sampling methods (§8).

4.2 Object creation: CMC on individual stems

A constructor by the same name as the class is used to create objects for this class. Constructors are available for both "standingTree" and "downLog" objects, as well as collections of either. Please see the help system for more details (`?crudeMonteCarlo`).

```
R> sTree = standingTree(dbh = 40, topDiam = 0, height = 20, solidType = 2.8)
R> sTree.cmc = crudeMonteCarlo(sTree, n.s = 40, segBnds = NA, startSeed = 124)
R> sTree.cmc
```

```
Object of class: crudeMonteCarlo
------------------------------------------------------------
crude Monte Carlo
```

```
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: cmcProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = 1.156013
Relative error % = 0.92435494
Variance estimate = 0.017320202
0.95% confidence Interval = 0.89 to 1.422
Number of samples n = 40
```

Recall that under the default taper function, which is used to create a "Stem" object, the `solidType` argument is the shape parameter with meaning $1 \leq$ `solidType` < 2 is a neiloid, 2 is a cone, and $(2, 10]$ is parabolic (Gove, 2011a).

4.3 CMC on stem collections

In general, one can easily make a collection of "Stem" subclass objects using the `downLogs` or `standingTrees` constructors (Gove, 2011a). In what follows, we will create each stem individually to illustrate how CMC works over a range of shape parameters for the taper model...

```
R> st.1 = standingTree(solidType = 1.2) #neiloid
R> st.2 = standingTree(solidType = 2)   #cone
R> st.3 = standingTree(solidType = 3)   #paraboloid
R> st.4 = standingTree(solidType = 4)   #paraboloid
R> st.5 = standingTree(solidType = 5)   #paraboloid
R> sTrees = standingTrees(list(st.1, st.2, st.3, st.4, st.5)) #collection
R> sapply(sTrees@trees, function(x) c(solidType = x@solidType,
+                                     topDiam = x@topDiam))
```

```
          tree.1 tree.2 tree.3 tree.4 tree.5
solidType    1.2      2      3      4      5
topDiam      0.0      0      0      0      0
```

```
R> sTrees.cmc = crudeMonteCarlo(sTrees, n.s = 40, startSeed = 283)
R> print(sTrees.cmc@stats, digits = 4)
```

```
              tree.1    tree.2    tree.3     tree.4     tree.5
trueVol    0.1497064 0.1902885 0.2295054  2.593e-01  0.2826739
```

```
volEst     0.1355428 0.2089213 0.2318706  2.326e-01  0.2803350
relErrPct -9.4609314 9.7918608 1.0305592 -1.031e+01 -0.8274368
volVar     0.0007536 0.0008015 0.0006832  5.058e-04  0.0005933
ci.lo      0.0800147 0.1516562 0.1790016  1.871e-01  0.2310650
ci.up      0.1910708 0.2661865 0.2847395  2.781e-01  0.3296049
```

Please see §11 for information on collections of "MonteCarlSampling" subclass objects.

5 The "importanceSampling" Class

Briefly, with importance sampling for bole volume (or some segment of the bole) one uses a proxy taper function from which to draw samples and thereby concentrate the samples in the lower portion of the bole, where there is more volume and measurements are easier. In general IS is a variance reduction method and is used in many other applications such as sequential Monte Carlo (Rubinstein and Kroese, 2008, p. 141, Gove, 2009).

Like CMC, the implementation of IS in `sampSurf` is not tied to any areal sampling method. It can be applied on any "Stem" subclass object (i.e., "downLog"s or "standingTree"s) independently of any areal method, so it can be used to do simple simulations on individual stems or collections thereof. Due to this flexibility, it can also be coupled with any areal method in `sampSurf` where applicable to estimate stem volumes in a two-stage design.

The "importanceSampling" class is a direct subclass of "crudeMonteCarlo" and adds no new slots to the definition. However, the estimators are different and this will be reflected in the quantities stored within the slots (Gregoire and Valentine, 2008, p. 110); *viz.*,

$$\hat{v}_s = G\frac{\rho_s}{g_s} \tag{5}$$

$$\hat{\bar{v}} = \frac{1}{n_s}\sum_{s=1}^{n_s}\hat{v}_s \tag{6}$$

The variance of the estimator (6) is unbiasedly estimated by (4) as under CMC. Sampled heights (h_s) depend on the proxy function chosen, and in general can be found by the method given in Gregoire and Valentine (2008, p. 109) for any proxy.

5.1 Class slots

There are not new slots with this class. Note, however, that the definitions for many of the slots in the CMC superclass will reflect the different estimators (5) and (6) given above.

5.2 Object creation: IS on individual stems

The following is a simple example to begin with using the default proxy (§8.3.2)...

```
R> sTree.is = importanceSampling(sTree, n.s = 40, segBnds = NA, startSeed = 124,
+                                proxy = 'gvProxy')
R> sTree.is
```

```
Object of class: importanceSampling
------------------------------------------------------------
Importance Sampling
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: gvProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = 1.1521355
Relative error % = 0.58582776
Variance estimate = 0.00093703462
0.95% confidence Interval = 1.09 to 1.214
Number of samples n = 40
```

These results can be compared directly with those in §4.2 for CMC.

5.3 IS on stem collections

Again, the following results can be compared with those for CMC in §4.3...

```
R> sTrees.is = importanceSampling(sTrees, n.s = 40, startSeed = 283)
R> print(sTrees.is@stats, digits = 4)
```

```
              tree.1    tree.2    tree.3    tree.4    tree.5
trueVol    0.1497064 0.1902885 0.2295054 2.593e-01 2.827e-01
volEst     0.1360497 0.1969432 0.2312196 2.593e-01 2.846e-01
relErrPct -9.1223071 3.4971451 0.7469406 2.141e-14 6.734e-01
volVar     0.0002569 0.0001092 0.0000216 5.531e-35 2.683e-05
ci.lo      0.1036302 0.1758071 0.2218198 2.593e-01 2.741e-01
ci.up      0.1684692 0.2180794 0.2406194 2.593e-01 2.951e-01
```

6 The "controlVariate" Class

Control variate sampling is sometimes referred to as difference sampling, and is also a variance reduction method (Rubinstein and Kroese, 2008, p. 123). The form implemented in `sampSurf` is that of Gregoire and Valentine (2008, p. 115), after Hammersley and Handscomb (1979, p. 59); *viz.*,

$$\begin{aligned}\hat{v}_s &= G + (b-a)\left(\rho(h_s) - g(h_s)\right) \\ &= G + (b-a)\delta_s \end{aligned} \tag{7}$$

for the sth sample point along the bole. The mean and the variance are given by the formulas in (6) and (4), respectively. Note that CMC is used to estimate the difference integral $\int_a^b \left(\rho(x) - g(x)\right) \mathrm{d}x$; therefore, the heights in (7) are drawn from $h_s = a + (b-a)u_s$, as in CMC.

6.1 Class slots

The "controlVariate" class is a subclass of "importanceSampling". The class slots are (including those from the "importanceSampling" superclass)...

```
R> showClass('controlVariate')
```

```
Class "controlVariate" [package "sampSurf"]

Slots:

Name:       diff.s       proxy      diam.s       rho.s       hgt.s       vol.s
Class:     numeric   character     numeric     numeric     numeric     numeric

Name:       volEst      volVar       ci.lo       ci.up  alphaLevel     trueVol
Class:     numeric     numeric     numeric     numeric     numeric     numeric

Name:    relErrPct        stem     segBnds         n.s   startSeed         u.s
Class:     numeric        Stem     numeric     numeric     numeric     numeric

Name:  description    userArgs
Class:   character        list

Extends:
Class "importanceSampling", directly
Class "crudeMonteCarlo", by class "importanceSampling", distance 2
Class "MonteCarloSampling", by class "importanceSampling", distance 3
```

where the only additional slot added in this class is defined as...

- *diff.s*: The differences $\delta_s = \rho(h_s) - g(h_s)$ in (7).

6.2 Object creation: CV on individual stems

The following is a simple example to begin with using the default proxy (§8.3.2)...

```
R> sTree.cv = controlVariate(sTree, n.s = 40, segBnds = NA, startSeed = 124,
+                            proxy = 'gvProxy')
R> sTree.cv


Object of class: controlVariate
------------------------------------------------------------
Control Variate Sampling
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: gvProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = -0.46944817
Relative error % = -140.98462
Variance estimate = 332.89781
0.95% confidence Interval = -37.4 to 36.44
Number of samples n = 40


R> sTree.cv@vol.s


 [1] -164.3360096  -35.1692420    7.1025794  -39.8962129 -108.9700415
 [6]  -81.3627868   34.4208090   -2.5750496  169.2710330  -86.3415230
[11]  109.0199424  142.9132457  103.6094391  140.3053357  -34.9700361
[16] -175.4274868   32.0598978   98.4155577  154.6499608 -185.0242773
[21]   50.6395505   42.1569947 -166.6191480  -33.1688431  -61.3903768
[26] -117.0980572  136.2928634  -91.7803956  132.6889992  -44.7383132
[31]  128.5241887  -75.2685469  -22.7850336  146.3211507 -153.3865767
[36]  188.0077116 -120.9616719  175.8197005 -138.3520425  -71.3752158
```

These results can be compared with those in the previous examples. Note, however, that employing '`gvProxy`' will result in a warning from the constructor that negative volumes have been estimated. These are shown in the last line above, and the resulting poor overall estimate must also be noted in the object summary. The reason for this outcome is the proxy itself, which is given in §8.3.2. This particular proxy is only *proportional* to cross-sectional area, and not closely so since it returns a height difference at the sth sample point along the stem, and not a cross-sectional area. Thus, the difference δ_s is one of two different quantities, the larger of which is the proxy quantity, and therefore produces the negative difference (and hence volume estimate).

When using control variate sampling, it is best to use a proxy that is close to that of the stem shape, and it is essential that it returns a value that is also proportionately close to the stem cross-sectional area at the sth point on the bole. A second proxy that is available in `sampSurf`, '`wbProxy`' (§8.3.3), is a better choice in this situation...

```
R> sTree.cv = controlVariate(sTree, n.s = 40, segBnds = NA, startSeed = 124,
+                            proxy = 'wbProxy', solidTypeProxy = 0.9)
R> sTree.cv
```

```
Object of class: controlVariate
------------------------------------------------------------
Control Variate Sampling
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: wbProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = 1.1557354
Relative error % = 0.90011644
Variance estimate = 8.870296e-06
0.95% confidence Interval = 1.15 to 1.162
Number of samples n = 40
```

```
R> any(sTree.cv@vol.s <= 0)
```

```
[1] FALSE
```

```
R> any(sTree.cv@diff.s <= 0)
```

```
[1] TRUE
```

In this case, we see that CV sampling does indeed produce reasonable results when an appropriate proxy is used (though in this case, the confidence interval failed to cover the true volume).

The above example uses a proxy taper shape parameter of $r = 2.52$, which is quite close to the true taper of 2.8. As we will see in §8.3.3, there is also another factor that comes into play with the default settings of this proxy: Because the tree tapers to a zero top diameter, the default in '`wbProxy`' is to change the tip to a larger diameter so that close to zero cross-sectional areas do not inflate IS volume estimates. However, this is not a concern in CV estimation, so we do not want the truncation, and can specify this as...

```
R> sTree.cv = controlVariate(sTree, n.s = 40, segBnds = NA, startSeed = 124,
+                            proxy = 'wbProxy', solidTypeProxy = NA,
+                            truncateProxyStem = FALSE)
R> sTree.cv
```

```
Object of class: controlVariate
------------------------------------------------------------
Control Variate Sampling
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: wbProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = 1.1454252
Relative error % = 0
Variance estimate = 2.5284003e-34
0.95% confidence Interval = 1.15 to 1.145
Number of samples n = 40
```

```
R> sum(sTree.cv@diff.s)
```

```
[1] -9.2807706e-17
```

The above run has used the exact taper (`solidTypeProxy = NA`) with no truncation of the proxy stem (`truncateProxyStem = FALSE`), producing δ_s values that are all zero.

6.3 CV on stem collections

Here we use the '`wbProxy`' function for the collection of stems to circumvent the large difference noted above...

```
R> sTrees.cv = controlVariate(sTrees, n.s = 40, startSeed = 283, proxy = 'wbProxy',
+                             solidTypeProxy = 0.9)
R> print(sTrees.cv@stats, digits = 4)
```

```
             tree.1    tree.2     tree.3     tree.4     tree.5
trueVol   1.497e-01 1.903e-01  2.295e-01  2.593e-01  2.827e-01
volEst    1.599e-01 1.986e-01  2.192e-01  2.491e-01  2.745e-01
relErrPct 6.830e+00 4.349e+00 -4.478e+00 -3.936e+00 -2.899e+00
volVar    2.284e-06 7.631e-07  9.505e-07  7.565e-07  9.210e-07
ci.lo     1.569e-01 1.968e-01  2.173e-01  2.474e-01  2.725e-01
ci.up     1.630e-01 2.003e-01  2.212e-01  2.509e-01  2.764e-01
```

7 The "antitheticSampling" Class

Under antithetic sampling, pairs of points are average for the sth estimate within the bole segment. The pairs are drawn using u_s and $u'_s = (1 - u_s)$ yielding sampling heights (lengths) h_s and h'_s, respectively. Antithetic sampling can be applied for any of the Monte Carlo methods described previously. The estimators for the sth sampling location on the bole are...

$$\hat{v}_s = \frac{(b-a)}{2}\left(\rho(h_s) + \rho(h'_s)\right) \qquad \text{CMC} \qquad (8)$$

$$\hat{v}_s = \frac{G}{2}\left(\frac{\rho(h_s)}{g(h_s)} + \frac{\rho_s(h'_s)}{g(h'_s)}\right) \qquad \text{IS} \qquad (9)$$

$$\hat{v}_s = G + \frac{(b-a)}{2}\left(\delta_s + \delta'_s\right) \qquad \text{CV} \qquad (10)$$

where the $\delta'_s = \rho(h'_s) - g(h'_s)$ are based on the u'_s uniform random numbers. The estimator for $n_s > 1$ and its variance are given by (6) and (4), respectively.

Application of antithetic sampling in `sampSurf` requires one first construct an object that is a subclass of "MonteCarloSampling". Then, simply, when the `antitheticSampling` constructor is applied, a new object of the same class is created on the u'_s random numbers from the original. This will be illustrated below.

7.1 Class slots

The "antitheticSampling" class is a separate class from the rest of the hierarchy given above. The reason for this is that it contains two objects of the "MonteCarloSampling" class hierarchy, and therefore is not a component in the hierarchy, but is more of a composite object. The class slots are...

```
R> showClass('antitheticSampling')
```

```
Class "antitheticSampling" [package "sampSurf"]

Slots:

Name:              mcsObj            mcsAnti             volEst
Class: MonteCarloSampling MonteCarloSampling            numeric

Name:              volVar              ci.lo              ci.up
Class:            numeric            numeric            numeric

Name:          alphaLevel            trueVol          relErrPct
Class:            numeric            numeric            numeric

Name:         description
Class:          character
```

where the new slots in this class are given as...

- *mcsObj*: This is the original "MonteCarloSampling" subclass object based on the u_s random numbers.
- *mcsAnti*: And this is the antithetic object companion to `mcsObj`, which is derived from the u'_s random numbers.

The remainder of the slots are defined exactly as those in the previous sections for "MonteCarloSampling" objects, using the definitions for the estimators given in (8)–(10).

7.2 Object creation: Antithetic sampling on individual stems

In what follows, antithetic sampling will be applied on each of the "MonteCarloSampling" subclass objects created above for individual stems.

7.2.1 Antithetic CMC

The results in the following can be compared directly with those in §4.2...

```
R> sTree.acmc = antitheticSampling(sTree.cmc)
R> summary(sTree.acmc, succinct = FALSE)


Object of class: antitheticSampling
------------------------------------------------------------
Antithetic Sampling
------------------------------------------------------------

Object of class: crudeMonteCarlo
------------------------------------------------------------
crude Monte Carlo
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: cmcProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = 1.156013
Relative error % = 0.92435494
Variance estimate = 0.017320202
0.95% confidence Interval = 0.89 to 1.422
Number of samples n = 40

Object of class: crudeMonteCarlo
------------------------------------------------------------
antithetic: crude Monte Carlo
------------------------------------------------------------
Original Stem object class: standingTree
Proxy taper function: cmcProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Volume estimate = 1.130719
Relative error % = -1.2839144
Variance estimate = 0.017954769
0.95% confidence Interval = 0.86 to 1.402
Number of samples n = 40
```

```
Antithetic estimates...
  Volume estimate = 1.143366
  Relative error % = -0.17977973
  Variance estimate = 0.00022865299
  0.95% confidence Interval = 1.113 to 1.174
```

Note in the above that both the estimate from the results in §4.2, plus the antithetic companion to it are shown, then the actual antithetic composite estimate is presented. This is a result of using `succinct = FALSE` in the `summary`, otherwise, only the antithetic estimate is presented as illustrated in the following sections.

7.2.2 Antithetic IS

Similarly for antithetic importance sampling—compare with §5.2...

```
R> sTree.ais = antitheticSampling(sTree.is)
R> sTree.ais
```

```
Object of class: antitheticSampling
------------------------------------------------------------
Antithetic Sampling
------------------------------------------------------------
Monte Carlo Class objects: importanceSampling
Original Stem object class: standingTree
Proxy taper function: gvProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Antithetic estimates...
  Volume estimate = 1.1506699
  Relative error % = 0.45787485
  Variance estimate = 6.2376915e-05
  0.95% confidence Interval = 1.135 to 1.167
  Number of samples n = 40
```

7.2.3 Antithetic CV

And finally, antithetic control variate sampling—compare with §6.2 (using '`wbProxy`')...

```
R> sTree.acv = antitheticSampling(sTree.cv)
R> sTree.acv
```

```
Object of class: antitheticSampling
------------------------------------------------------------
Antithetic Sampling
------------------------------------------------------------
Monte Carlo Class objects: controlVariate
Original Stem object class: standingTree
Proxy taper function: wbProxy
Full tree height = 20
Segment height bounds = 0 to 20
True volume = 1.1454252
Antithetic estimates...
  Volume estimate = 1.1454252
  Relative error % = 0
  Variance estimate = 0
  0.95% confidence Interval = 1.145 to 1.145
  Number of samples n = 40
```

7.3 Antithetic sampling on stem collections

As we have seen, antithetic sampling can be applied to any one of the other Monte Carlo methods. Therefore, to construct a collection of "antitheticSampling" objects, we simply use a collection from one of the other Monte Carlo sampling methods. For example, the most common use is perhaps to apply antithetic sampling in conjunction with crude Monte Carlo; *viz.*,

```
R> sTrees.anti = antitheticSampling(sTrees.cmc)
R> print(sTrees.anti@stats, digits = 4)
```

```
             tree.1    tree.2    tree.3    tree.4     tree.5
trueVol   0.1497064 1.903e-01 2.295e-01 2.593e-01  2.827e-01
volEst    0.1611046 1.927e-01 2.296e-01 2.593e-01  2.812e-01
relErrPct 7.6137084 1.289e+00 4.353e-02 2.141e-14 -5.179e-01
volVar    0.0001338 3.673e-05 4.883e-06 2.963e-35  2.102e-06
ci.lo     0.1377086 1.805e-01 2.251e-01 2.593e-01  2.783e-01
ci.up     0.1845006 2.050e-01 2.341e-01 2.593e-01  2.841e-01
```

Comparing these results with those in §4.3 we see that antithetic sampling has improved the estimates in all cases.

8 Proxy Functions

All of the above constructor functions that create the different classes of objects for the various Monte Carlo sampling methods ultimately call the base `importanceSampling` constructor. Its signature argument is a `list` object that contains the information for the "Stem" object (or objects in a collection).[1] The proxy function is passed as a character argument denoting the name of the function (see examples in previous and following sections). The function itself is then retrieved from either the `sampSurf` package namespace (if it is one of those listed below), or from the search path. Then it is checked for conformance in argument names and return values before being applied in the desired sampling method. All this is accomplished by the `getProxy` function.[2]

The following could be a little confusing. The term "proxy function" is used in three different contexts. First, it can refer to the actual R function passed as an argument to the desired Monte Carlo constructor, as noted in the previous paragraph. Secondly, there is the mathematical proxy function, g_s, which we have encountered previously. Finally, g_s needs its own R function so it can be evaluated within the constructors methods. We will see how this is done in the following sections. Hopefully the distinction will be clear enough as to what "proxy function" we are referring to in what follows without having to restate it each time.

8.1 Using proxy functions

The `sampSurf` package provides some built-in proxy functions for use in Monte Carlo sampling within trees or logs. These are detailed below. However, one can write one's own proxy functions and use them within the system, as long as they meet the specifications in the next section. The R code for the proxies provided with `sampSurf` are a good place to start when planning to write one's own method. They go from very simple—e.g., in the CMC proxy (§8.3.1)—to fairly complex, in '`wbProxy`' which evaluates the default taper proxy (§8.3.3).

8.2 The format of proxy functions

Proxy functions (e.g., §8.3.1–8.3.3) should always have the following argument list at a minimum...

1. `stem`: A valid "Stem" object.
2. `u.s`: The uniform random number vector at which sample heights will be calculated.
3. `segBnds`: The segment bounds/limits, as previously described—length two.

[1] See `methods?importanceSampling` for more information.
[2] See `?getProxy` for more information.

4. `...`: The normal dotted argument list in R.

The `getProxy` function simply checks that the first three arguments above are indeed the first three in the proxy argument list, and that the dotted list is also present. Note that this allows one to define proxies with other named arguments in addition to the above (e.g., '`wbProxy`' below), but the above must be there too.

All proxy functions should return a `list` with the following named components...

1. `g`: a `closure` (function) with one argument, `hgt`, for the height or length along the stem at which cross-sectional area is returned. This R function corresponds to the actual mathematical proxy function g_s that has been described in the previous sections on Monte Carlo sampling methods, and will be discussed in more detail below.
2. `G`: a scalar, the volume integral of the g_s proxy within `segBnds`.
3. `hgt.s`: the sampled heights, h_s, along the bole at which the proxy function, `g`, will be evaluated.

These names are checked for correspondence in the `return` statement of the function body. Please note that if you decide to write a proxy function for the system other than those described below, it is up to you to make sure these return values are of the correct type desired for correct functionality of the sampling system. Please see the functions below for examples.

Please also note that the returned proxy function `g` is a *closure*. Therefore, it contains the environment of the function it was defined within (i.e., the function specified by the character name one passes in the `proxy` argument to, e.g., the `importanceSampling` constructor). As such, R will look within this enclosing environment for undefined objects before searching elsewhere. The '`wbProxy`' is an example of how this environment is used (and indeed is extremely useful) to define the proxy "Stem" object and keep it for evaluation of the built-in default taper function. Please see Chambers (2008, p. 125) and Gentleman (2008, p. 59) for good explanations of lexical scoping and closures in R.

8.3 Built-in proxies

`sampSurf` has three proxies that are available for use in Monte Carlo sampling. They are presented below in order from the simplest to the most complex.

8.3.1 cmcProxy

The first is the simple CMC proxy '`cmcProxy`'. This is employed in CMC sampling and in estimating the difference integral for CV sampling.

In the following example, we get the CMC proxy function, using `getProxy` and apply it to the "standingTree" object created previously...

```
R> cmcFun = getProxy('cmcProxy')
R> c( class(cmcFun), typeof(cmcFun) )
```

```
[1] "function" "closure"
```

```
R> cmcFun
```

```
function (stem, u.s, segBnds, ...)
{
    g = function(hgt = NA) 1/(segBnds[2] - segBnds[1])
    G = 1
    hgt.s = segBnds[1] + u.s/g()
    return(list(g = g, G = G, hgt.s = hgt.s))
}
<environment: namespace:sampSurf>
```

```
R> ( cmcFun(sTree, runif(4), c(0, sTree@height)) )
```

```
$g
function (hgt = NA)
1/(segBnds[2] - segBnds[1])
<environment: 0xa8d3778>

$G
[1] 1

$hgt.s
[1] 4.4851675 6.3536959 5.7916160 3.2573259
```

In the above the `getProxy` function has been used to retrieve the `sampSurf` implementation of a CMC proxy. Note that the return value is of the form discussed in the previous section, and indeed

if it were not, would have resulted in an error. Be especially mindful of the R environment returned with the `g` function component of the `list`. As noted above, `g` is the actual g_s mathematical uniform proxy in R code, and is a closure with the enclosing environment—that of '`cmcProxy`'—shown. Note also that, even though the `hgt` argument to `g` is irrelevant in this case, it is still defined, though set to `NA` as a default since it is never used.

8.3.2 gvProxy

By default (`proxy = 'gvProxy'`), the proxy function used in the `importanceSampling` constructor is that of Gregoire and Valentine (2008, p. 111). It is a very simple proxy: $g(h_s) = H - h_s$, where g_s is, in this case, proportional to cross-sectional area at height h_s. The other necessary components for IS based on this proxy are given by those authors.

```
R> args(getProxy('gvProxy'))
```

```
function (stem, u.s, segBnds, ...)
NULL
```

As noted in §6.2, this proxy can cause negative volumes under CV sampling. The reason for this is because $g(h)$ is only proportional to cross-sectional area, and returns a height difference that is on a different scale than cross-sectional area (normally larger unless h is close to H), causing a negative difference in the CV estimate formula (7). In addition, it is possible to generate large estimates under IS when this proxy is used with stems that do not taper to the tip. The reasons, and an example, are given in §12.2.2.

8.3.3 wbProxy

A third proxy (`proxy = 'wbProxy'`) is based on the default taper equation system used in `sampSurf`. This taper system is discussed in detail in Gove (2011a), but a little review will be given here. The following are the default diameter taper and volume equations, respectively...

$$d(l) = D_u + (D_b - D_u)\left(\frac{L-l}{L}\right)^{\frac{2}{r}} \tag{11}$$

$$v(l) = \frac{\pi}{4}\left[D_u^2 l + L(D_b - D_u)^2 \frac{r}{r+4}\left(1 - \left(1 - \frac{l}{L}\right)^{\frac{r+4}{r}}\right)\right.$$
$$\left. + 2LD_u(D_b - D_u)\frac{r}{r+2}\left(1 - \left(1 - \frac{l}{L}\right)^{\frac{r+2}{r}}\right)\right] \tag{12}$$

where D_b is the large-end or butt diameter, with small-end diameter D_u, $0 \leq l \leq L$ is the intermediate log length (or intermediate tree height, h) for volume or diameter estimates, L is log length (tree height, H), and r is a shape parameter such that $0 \leq r < 2$ generates a neiloid, $r = 2$ generates a cone, and $r > 2$ generates a paraboloid with limits imposed in `sampSurf` of `c(1, 10)`. The r parameter is specified directly via the `solidType` argument when constructing a "Stem" subclass object as mentioned earlier. Note that with these taper and volume equations, the units for diameters must be the same as for length. Cross-sectional area for the proxy is simply $g(h_s) = \pi d(h_s)^2/4$ from (11) squared. Similarly, using (12) the proxy volume is simply $G \equiv v(H)$.

```
R> args(getProxy('wbProxy'))
```

```
function (stem, u.s, segBnds, solidTypeProxy = 3, truncateProxyStem = TRUE,
    wbProxySolve = c("uniroot", "nlminb"), warningsOn = FALSE,
    ...)
NULL
```

The '`wbProxy`' function contains a proxy "Stem" object that can take on any of the shapes specified by the shape parameter, which is explicitly specified through the `solidTypeProxy` argument. In addition, one can use `solidTypeProxy` $\in (0, 1)$ to specify a proportion of the true `solidType` (r parameter) for the log or tree being sampled such that the proxy shape is then `max(1, solidTypeProxy`$\times$`solidType)`. Thus, for example, specifying `solidTypeProxy = 0.99` will make the proxy stem almost exactly the same taper as the log being sampled. Finally, if `solidTypeProxy = NA` is specified, then the exact taper for the log or tree is used for the proxy stem, which should lead to zero relative error in IS as long as the stem does not taper to zero diameter at the tip.

Under importance sampling, it is possible for the estimator to inflate when the proxy cross-sectional area goes to zero on a stem that tapers to the tip (`topDiam` of zero), since g_s appears in the denominator of (5). The '`wbProxy`' function allows one to choose whether or not to truncate the proxy taper function for importance sampling, thereby eliminating potential bias due to inflation[3]. By calling '`wbProxy`' with argument `truncateProxyStem = TRUE` (default), the proxy stem is truncated (actually, the tip diameter is enlarged, keeping the same stem length/height) at about one inch (2.54 cm for metric) so that any individual estimates drawn from heights near the tip do not inflate unreasonably.[4] In this case, there will be a small amount of error in the IS estimate with the exact taper model as proxy (i.e., when `solidTypeProxy = NA`) because of the truncation; that is, the exact taper and the proxy taper no longer match exactly due to enlargement of `topDiam` in the proxy stem. Under CV sampling, the estimator (7) does not have this potential inflation problem, and it is better to have the proxy stem be as close to the shape of the original tree as possible, so it would be preferable to call '`wbProxy`' with `truncateProxyStem = FALSE`. Truncating

[3] Note that in order for a bias to accrue due to inflation, a cross-sectional area near the tip needs to be sampled; this is more likely on stems with large `solidType` values

[4] Truncation only happens on proxy stems that taper to `stem@topDiam<0.01` (for both English and metric); proxy stems with `topDiam` larger than this are unaffected.

the stem under CV will actually lead to a bias on stems that taper to the tip because the stem taper and proxy taper no longer match. Somewhat paradoxically, truncation can slightly improve CV estimation when the actual stem has a top diameter somewhere between zero and one inch. One can experiment with the various settings and verify these assertions; however, it should be kept in mind that the above points will also depend on the shape of the proxy stem defined through the `solidTypeProxy` argument and the actual stem shape (`solidType`).[5]

In the case where the stem taper has been supplied by the user and not the default taper equation (i.e., `stem@solidType=NULL`), either through measurements or through a taper equation, the above model can still be applied as a proxy. If `solidTypeProxy` is in the range allowed by `sampSurf` then that shape model is used. If it is specified as `NA`, then a spline is fitted to the taper data for both the proxy and for the bole segment volume; again, this will make the IS estimate exact as above. However, specifying `solidTypeProxy` $\in (0, 1)$, does not make sense in this case and will result in using a value of $r = 3$ for the proxy function as a compromise.

In the examples already presented, the '`gvProxy`' function was used. Below we look at a few options using the '`wbProxy`' for comparison. First, we specify a parabolic proxy taper for all trees...

```
R> sTrees.is = importanceSampling(sTrees, n.s = 40, startSeed = 283,
+                                 proxy = 'wbProxy', solidTypeProxy = 3)
R> print(sTrees.is@stats, digits = 6)
```

```
               tree.1       tree.2       tree.3      tree.4      tree.5
trueVol    1.49706e-01  1.90289e-01  2.29505e-01 2.59333e-01  0.28267392
volEst     1.10842e-01  1.63425e-01  2.14950e-01 2.62767e-01  0.26930297
relErrPct -2.59605e+01 -1.41170e+01 -6.34226e+00 1.32420e+00 -4.73016552
volVar     1.25398e-05  9.41666e-06  3.66971e-06 9.54835e-05  0.00043293
ci.lo      1.03679e-01  1.57219e-01  2.11075e-01 2.43002e-01  0.22721691
ci.up      1.18005e-01  1.69632e-01  2.18824e-01 2.82532e-01  0.31138904
```

Notice the fairly substantial error associated with the first two trees for this example; e.g., the relative error for tree one (`solidType = 1.2`)is -25.96%. Also note that the third tree would have zero error if the tree did not taper to the tip, since its shape parameter is the same as the proxy. As another example,

```
R> sTrees2 = standingTrees(5, topDiams = c(0.2,0.3))
R> sapply(sTrees2@trees, function(x) c(solidType = x@solidType,
+                                       topDiam = x@topDiam))
```

[5] In the future, `truncateProxyStem` may become a numeric— truncation diameter—instead of a logical, this would allow experimentation with the truncation effect.

```
          tree.1 tree.2 tree.3 tree.4 tree.5
solidType 6.7000 6.7000 1.3000 7.1000 4.7000
topDiam   0.0353 0.0982 0.0383 0.0277 0.0875
```

```
R> sTrees2.is = importanceSampling(sTrees2, n.s = 40, startSeed = 283,
+                                  proxy = 'wbProxy', solidTypeProxy = NA)
R> print(sTrees2.is@stats, digits = 4)
```

```
             tree.1    tree.2    tree.3    tree.4    tree.5
trueVol   8.840e-02 6.106e-01 1.036e-01 7.003e-02 5.966e-01
volEst    8.840e-02 6.106e-01 1.036e-01 7.003e-02 5.966e-01
relErrPct 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
volVar    3.827e-36 1.264e-34 4.568e-36 1.852e-36 9.482e-35
ci.lo     8.840e-02 6.106e-01 1.036e-01 7.003e-02 5.966e-01
ci.up     8.840e-02 6.106e-01 1.036e-01 7.003e-02 5.966e-01
```

Here we have selected new trees with non-zero top diameter. The option for `solidTypeProxy = NA` was used, specifying that the proxy exactly equal the true taper model for each bole. The result, as expected, are IS estimates that are perfect.

The following demonstrates CV sampling on a range of tree shapes with no truncation in the proxy. The proxy stem again has a parabolic shape...

```
R> sapply(sTrees@trees, function(x) c(solidType = x@solidType,
+                                      topDiam = x@topDiam))
```

```
          tree.1 tree.2 tree.3 tree.4 tree.5
solidType    1.2      2      3      4      5
topDiam      0.0      0      0      0      0
```

```
R> sTrees.cv = controlVariate(sTrees, n.s = 40, startSeed = 283,
+                             proxy = 'wbProxy', solidTypeProxy = 3,
+                             truncateProxyStem = FALSE)
R> print(sTrees.cv@stats, digits = 6)
```

```
                tree.1       tree.2      tree.3      tree.4      tree.5
trueVol    1.49706e-01  1.90289e-01 2.29505e-01 2.59333e-01 2.82674e-01
volEst     3.33615e-02  1.37236e-01 2.29505e-01 2.99100e-01 3.42629e-01
relErrPct -7.77154e+01 -2.78798e+01 0.00000e+00 1.53343e+01 2.12098e+01
```

```
volVar     1.18395e-04  1.42754e-05 1.53087e-35 6.41596e-06 1.83786e-05
ci.lo      1.13526e-02  1.29594e-01 2.29505e-01 2.93976e-01 3.33957e-01
ci.up      5.53703e-02  1.44879e-01 2.29505e-01 3.04223e-01 3.51300e-01
```

This example illustrates that the closer the actual shape of the stem is to the specified proxy, the better CV is (the smaller the difference adjustment). The third tree matches the proxy exactly, as we depart from the ideal shape in either direction (cone to neiloid or "heavier" paraboloids), the differences, and thus the error in CV sampling, increases accordingly. This example shows that the errors can become quite large when the stem bole and proxy are severely mismatched in shape.

8.3.4 Finding the proxy "Stem" object

The following assumes that one is familiar with *closures* in R.[6] There is nothing mysterious about closures, but many R users may have never delved into the subject. To help, a small introduction to closures can be found in Appendix A, which may be helpful to those who want to write their own proxy functions. It should also help to take any mystery out of what follows if one is not familiar with closures in R.

The '`wbProxy`' function utilizes closures to keep the proxy stem object around so that it can be used in the call to cross-sectional area computation via the returned `g` function. The closure for the `g` function is the environment for the `wbProxy` function. The following illustrates how to get a copy of the proxy stem and work a little with it if desired...

```
R> wbp = getProxy('wbProxy')
R> (pr = wbp(sTree, runif(1), c(0, sTree@height)))


$g
function (hgt)
csaFactor * taperInterpolate(stem, "diameter", hgt)^2
<environment: 0xae6eb68>

$G
[1] 1.1454252

$hgt.s
[1] 1.9668057


R> pTree = get('stem', envir=environment(pr$g))    #proxy stem
R> ls(envir = environment(pr$g))
```

[6] Please feel free to skip this section, it is here for completeness in documentation, and understanding it is not required for use of the '`wbProxy`' proxy—see the references cited in §8.2 and Appendix A for more on closures.

```
 [1] "csaFactor"         "fh.lower"          "fh.nlm"
 [4] "fh.uni"            "fh.upper"          "g"
 [7] "G"                 "hgt.s"             "i"
[10] "n.u"               "nt"                "penulDiam"
[13] "segBnds"           "solidTypeProxy"    "stem"
[16] "truncateProxyStem" "truncDiam"         "u.s"
[19] "units"             "warningsOn"        "wbProxySolve"
```

```
R> tail(cbind(sTree@taper, pTree@taper))
```

```
      diameter height    diameter height
16 0.156334931     15 0.156334931     15
17 0.133301363     16 0.133301363     16
18 0.108540704     17 0.108540704     17
19 0.081248091     18 0.081248091     18
20 0.049521266     19 0.049521266     19
21 0.000000000     20 0.025400000     20
```

In the first line we retrieve the '`wbProxy`' function from the `sampSurf` namespace, while checking that it is of the correct format. Then this function is applied in the second line to a single "standingTree" object, using one random height, showing both the call structure and the return values discussed in §8.2. The third line then retrieves a copy of the proxy stem object from the closure for `g` (note that `g` was returned in the `list` from the function call in the previous line). The fourth line shows a listing of the objects found within the environment (note that `stem`, the proxy stem, is one of the objects). Finally, we compare the last few segments of the taper for both the true stem and the proxy. Notice that the proxy stem has been "truncated" or enlarged at the tip (`topDiam`) for reasons previously discussed.

Thus, once we have made a call to the '`wbProxy`' function (here, via `wbp`) for a given "Stem" subclass object, the proxy stem object will always be available in this manner. This is how the function `g`, as shown in the above code snippet, is able to find the `stem` object (the proxy stem) in the call to `taperInterpolate`. For example, the following backs out the diameter from the cross-sectional area returned from a call to `g`, at the sampled height `pr$hgt.s = 1.97`...

```
R> sqrt(pr$g(pr$hgt.s)/.StemEnv$baFactor['metric'])
```

```
    metric
0.39445385
```

```
R> iph = findInterval(pr$hgt.s, pTree@taper[,'height'])
R> pTree@taper[seq(iph-1, iph+1, 1), ]
```

```
    diameter height
1 0.42082243      0
2 0.40568333      1
3 0.39031474      2
```

We see that the correct cross-sectional area has been interpolated from the proxy taper. Note that no reference to `stem` is required, because it is automatically found in `g`'s enclosing environment.

8.4 A few more extensive examples

Here we simply look at a few more examples using the built-in proxy functions. For a change, we will use a set of logs using the '`gvProxy`' function...

```
R> dLogs = downLogs(20, solidTypes = c(2,8))
R> dLogs.is = importanceSampling(dLogs, n.s = 100, startSeed = 142,
+                                proxy = 'gvProxy')
R> summary(dLogs.is@stats['relErrPct',])
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-12.51000  -1.48240   0.34101   1.69490   2.25730  37.32100
```

Next, the same population of logs with varying taper proxy models; first is conical...

```
R> dLogs.is = importanceSampling(dLogs, n.s = 100, startSeed = 142,
+                                proxy = 'wbProxy', solidTypeProxy = 2)
R> summary(dLogs.is@stats['relErrPct',])
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
 0.44608  8.87550 21.25300 22.81500 37.11000 53.05000
```

The next two are varying degrees of parabolic proxy taper...

```
R> dLogs.is = importanceSampling(dLogs, n.s = 100, startSeed = 142,
+                                proxy = 'wbProxy', solidTypeProxy = 3)
R> summary(dLogs.is@stats['relErrPct',])
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-11.5560   2.6394  12.7360  11.9680  20.9530  35.9510
```

```
R> dLogs.is = importanceSampling(dLogs, n.s = 100, startSeed = 142,
+                                proxy = 'wbProxy', solidTypeProxy = 5)
R> summary(dLogs.is@stats['relErrPct',])
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-21.56300  -4.81800  -1.76740  -0.40944   5.34770  12.10000
```

Finally, we let the proxy taper equation be close to the true taper...

```
R> dLogs.is = importanceSampling(dLogs, n.s = 100, startSeed = 142,
+                                proxy = 'wbProxy', solidTypeProxy = 0.9)
R> summary(dLogs.is@stats['relErrPct',])
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.3188  1.3566  2.2252  2.3513  3.6555  5.8575
```

The results corroborate the previous examples, illustrating that the shape of the proxy compared to the actual stem will influence the results. This also holds for CV sampling, and can easily be verified with similar simulations.

9 Estimating Segment Volumes

The methods allow the capability of estimating segment or bolt volumes that are portions of the entire bole. For example, in the following, first we estimate whole log volume, then the volume of a segment...

```
R> dLog = downLog(buttDiam = 50, topDiam = 10, logLen = 8, solidType = 3,
+                 nSegs = 50)
R> dLog.is = importanceSampling(dLog, n.s = 20, startSeed = 390)
R> dLog.is
```

```
Object of class: importanceSampling
------------------------------------------------------------
```

```
Importance Sampling
------------------------------------------------------------
Original Stem object class: downLog
Proxy taper function: gvProxy
Full log length = 8
Segment length bounds = 0 to 8
True volume = 0.79527174
Volume estimate = 0.78671527
Relative error % = -1.0759182
Variance estimate = 1.38302e-05
0.95% confidence Interval = 0.779 to 0.7945
Number of samples n = 20
```

```
R> dLogSeg.is = importanceSampling(dLog, n.s = 20, segBnds = c(2,5),
+                                  startSeed = 390, proxy = 'gvProxy')
R> dLogSeg.is
```

```
Object of class: importanceSampling
------------------------------------------------------------
Importance Sampling
------------------------------------------------------------
Original Stem object class: downLog
Proxy taper function: gvProxy
Full log length = 8
Segment length bounds = 2 to 5
True volume = 0.32796803
Volume estimate = 0.32812888
Relative error % = 0.049044499
Variance estimate = 2.15132e-07
0.95% confidence Interval = 0.327 to 0.3291
Number of samples n = 20
```

The segment height bounds, of course, must lie within the bole.

Segment estimates can also be made for a collection of stems, but one must be careful that the segment lies within the bole of *every* stem in the collection in this case. If it is desired to have segments that differ by stem, simply process the stems individually and then summarize them as desired. For example, here we collectively process with a common segment...

```
R> sapply(sTrees@trees, function(x) x@height)
```

```
tree.1 tree.2 tree.3 tree.4 tree.5
    15     15     15     15     15
```

```
R> sTreesSeg.is = importanceSampling(sTrees, n.s = 40, segBnds = c(5,10),
+                                    startSeed = 283)
R> print(sTreesSeg.is@stats, digits = 4)
```

```
              tree.1    tree.2    tree.3    tree.4    tree.5
trueVol    2.455e-02 4.933e-02 7.143e-02 8.644e-02 9.712e-02
volEst     2.312e-02 5.018e-02 7.151e-02 8.644e-02 9.727e-02
relErrPct -5.817e+00 1.717e+00 1.108e-01 1.605e-14 1.534e-01
volVar     2.690e-06 2.101e-06 4.973e-07 6.296e-36 4.283e-07
ci.lo      1.981e-02 4.725e-02 7.008e-02 8.644e-02 9.594e-02
ci.up      2.644e-02 5.311e-02 7.293e-02 8.644e-02 9.859e-02
```

10 Plotting "MonteCarloSampling" Objects

There is a `plot` method for objects that are a subclass of "MonteCarloSampling" that allows one to visually represent the Monte Carlo sampling points along the bole. The plots are fairly rudimentary, but perhaps helpful. Figure 2 presents an example for a "downLog" with the '`gvProxy`' using the default `renderAs = 'profile'`.

```
R> opar = par(mfrow = c(1,2))
R> plot(dLog.is, axes=TRUE)
R> plot(dLogSeg.is, axes=TRUE)
```

Note that most of the IS points are relatively low on the bole when sampling the entire stem in this example. The second plot in this figure shows IS when restricted to a segment of the stem.

An alternative presentation is to show the same information in cross-section view...

```
R> plot(dLog.is, renderAs = 'crossSection')
R> axis(1)
R> plot(dLogSeg.is, renderAs = 'crossSection')
R> axis(1)
```

The cross-section plot is shown in Figure 3, and can be compared directly with the profile plot in Figure 2.

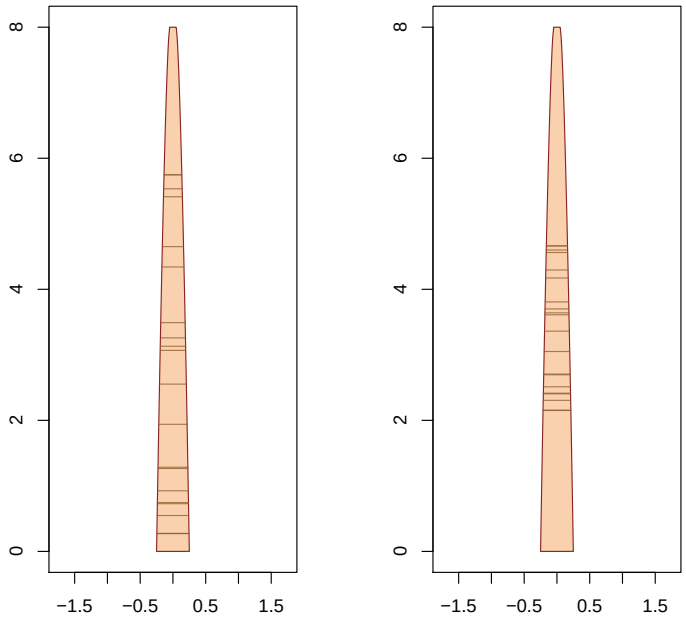

Figure 2: Default profile plots for object of class "importanceSampling", illustrating the sample points chosen along the entire stem (left) and a segment of the stem (right).

In either case, the stem is presented at the origin rather than at its actual location to facilitate accurate dimensional comparison along the bole and cross-section.

11 Container Classes for collections

The now familiar "container" classes are used to store and aid in the manipulation of collections of objects that are subclasses of "MonteCarloSampling", or an object of class "antitheticSampling". The examples in the previous sections have illustrated how simple it is to create collections of objects, but there have been no details given heretofore on the structure of these objects, or the methods available to view them. In general summary statistics for all stems in the collection are calculated when the collection is created. Furthermore, collections should only contain one type of sampling to be valid; e.g., control variate sampling applied to all "Stem" objects passed to the container constructor.

Please note that the correct way to create collections is by using the constructors given in the

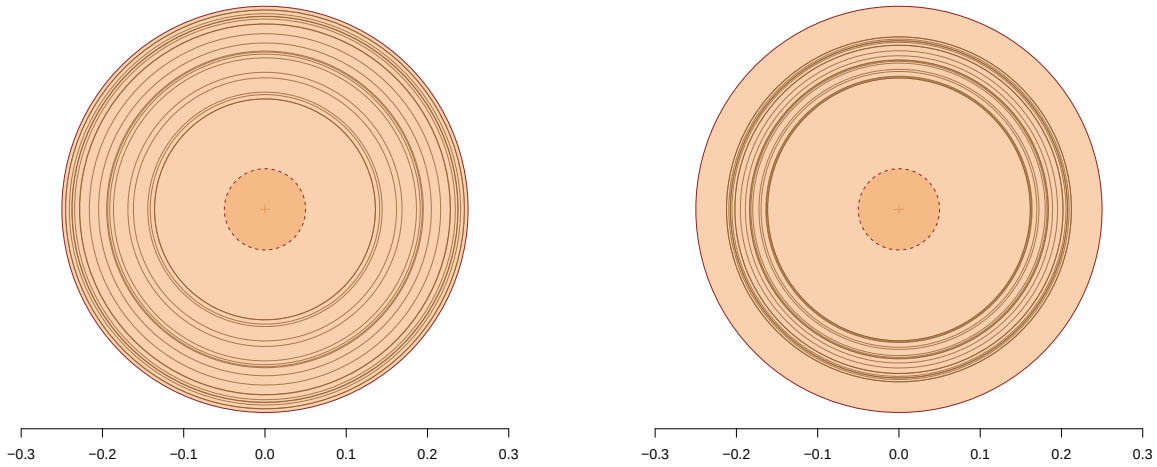

Figure 3: Default cross-section plots for object of class "importanceSampling", illustrating the sample points chosen along the entire stem (left) and a segment of the stem (right). Note that the top diameter is displayed as dashed (if non-zero).

examples in the previous and following sections. Use of the actual constructors that are named after the classes below (i.e., "mcsContainer" and "antitheticContainer") is not required, and is discouraged. This is a slightly different approach than is found elsewhere in `sampSurf`, but is a simpler way to proceed in this case.

11.1 The "mcsContainer" class

This class stores collections of objects that are a subclass of "MonteCarloSampling".

```
R> showClass('mcsContainer')
```

```
Class "mcsContainer" [package "sampSurf"]

Slots:

Name:      mcsObjs       stats description
Class:        list      matrix   character

Known Subclasses: "antitheticContainer"
```

11.2 Class slots

- *mcsObjs*: A `list` containing objects that are each a common subclass of "MonteCarloSampling". Note that mixing sampling methods within this list will result in an invalid object.
- *stats*: A matrix with the summary for the main components of the individual class objects.
- *description*: A `character` description of the object if desired (defaults are given for the class).

11.3 The "antitheticContainer" class

This class stores collections of objects that are a subclass of "antitheticSampling".

```
R> showClass('antitheticContainer')
```

```
Class "antitheticContainer" [package "sampSurf"]

Slots:

Name:      mcsObjs       stats description
Class:        list      matrix   character

Extends: "mcsContainer"
```

Note that this is a subclass of "mcsContainer", with the same slots. The difference is that only objects of class "antitheticSampling" can be held in the `mcsObjs list` slot.

11.4 Uses of collections

We have actually already used these classes without knowing it (sometimes it is better not to know). For example, in §4.3 we created a collection of "crudeMonteCarlo" objects and looked at the `stats` slot. We repeat that below, and then look at a `summary` on the same object...

```
R> print(sTrees.cmc@stats, digits = 4)
```

```
              tree.1    tree.2    tree.3     tree.4     tree.5
trueVol    0.1497064 0.1902885 0.2295054  2.593e-01  0.2826739
```

```
volEst     0.1355428 0.2089213 0.2318706  2.326e-01  0.2803350
relErrPct -9.4609314 9.7918608 1.0305592 -1.031e+01 -0.8274368
volVar     0.0007536 0.0008015 0.0006832  5.058e-04  0.0005933
ci.lo      0.0800147 0.1516562 0.1790016  1.871e-01  0.2310650
ci.up      0.1910708 0.2661865 0.2847395  2.781e-01  0.3296049
```

```
R> sTrees.cmc
```

```
------------------------------------------------------------
Monte Carlo Sampling container object
------------------------------------------------------------
There are 5 crudeMonteCarlo objects in the collection

Summary stats over all objects...
        trueVol  volEst relErrPct     volVar    ci.lo   ci.up
Min.    0.14971 0.13554 -10.30900 0.00050578 0.080015 0.19107
1st Qu. 0.19029 0.20892  -9.46090 0.00059334 0.151660 0.26619
Median  0.22951 0.23187  -0.82744 0.00068319 0.179000 0.27809
Mean    0.22230 0.21785  -1.95490 0.00066750 0.165770 0.26994
3rd Qu. 0.25933 0.23260   1.03060 0.00075364 0.187110 0.28474
Max.    0.28267 0.28033   9.79190 0.00080153 0.231070 0.32960

Proxy tabulation...
cmcProxy
       5
```

The `summary` shows the number of "crudeMonteCarlo" objects in the collection, which corresponds to the number of trees. Then it gives a summary over the stats for all trees in the collection. Finally, it shows a tabular breakdown of the proxy functions used. Normally, they will all be of one type, but there is no restriction on this (except in "crudeMonteCarlo" objects the built-in '`cmcProxy`' is always used), one can mix proxies among trees in the collection if desired.

11.4.1 Plotting the object

Limited graphics are available for the container object collections. One can use `hist` to display a histogram of either the `relErrPct` or `volVar` slots across stems in the collection. In addition, the `plot` method can be used to look at plots of `trueVol` versus `volEst` for each technique. Of course, one is free to produce any other plots desired with a little coding, these are just the built-in choices when either function is applied to a container object; e.g.,

```
R> plot(dLogs.is, xlab = 'Estimated Volume', ylab = 'True volume', cex = 1.25,
+      cex.lab = 1.25)
R> hist(dLogs.is)
```

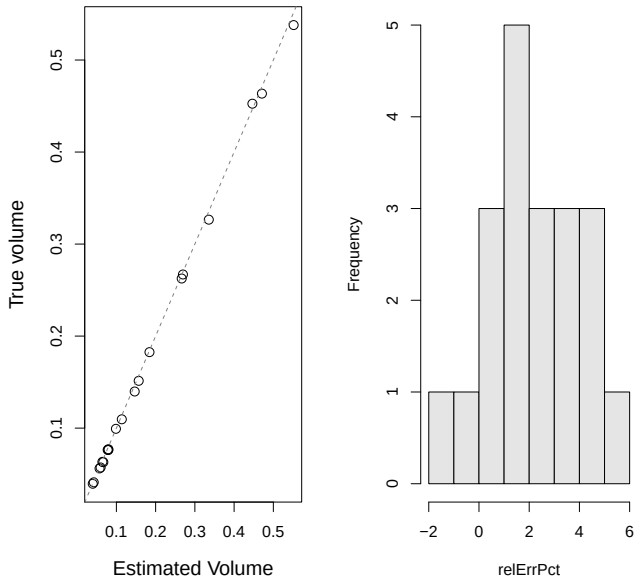

Figure 4: A scatter plot of the volumes in a collection and associated histogram of relative errors.

The results shown in Figure 4 are self-explanatory.

12 Monte Carlo Sampling within Areal Methods

The discussion to this point has centered strictly on using Monte Carlo sampling methods within "downLog" or "standingTree" objects to estimate volume. This application is independent of the normal areal sampling frame approach in `sampSurf` and, as previously mentioned, can thus be used to assess the efficacy of Monte Carlo methods on collections of trees or logs within stem-based simulation experiments. Much of the previous work on these methods has centered on this approach, though some studies (e.g., Gregoire et al., 1986; Valentine et al., 1990) have described the two-stage approach in estimating volumes from an inventory where stem-based Monte Carlo variants are used with areal methods like fixed-area plot sampling. Interestingly though, neither field nor simulation studies on the two-stage approach have been undertaken (to the author's knowledge).

Of course, there are already a number of methods in `sampSurf` that are based on Monte Carlo

principals. For example, any of the PDS variants as well as the Monte Carlo variant of distance-limited sampling (DLMCS) for downed logs use this idea; and critical height sampling is a crude Monte Carlo approach on standing trees. In these approaches, there is a designed constraint in the sampling point, producing sampling surfaces that have a predictable shape that is reproducible and not random. In what follows, however, the Monte Carlo methods are applied in a two-stage manner without any such constraint, yielding random sampling surfaces that will change from one realization to the next (unless the random seed is used to replicate a surface).

First, a tract object is required now that we are doing areal methods. In the following we establish the buffered tract, within which a synthetic tree population is created...

```
R> smtr = Tract(c(x = 70, y = 70), cellSize = 1)
R> smbtr  = bufferedTract(10, smtr)
R> sTrees3 = standingTrees(6, smbtr, dbhs = c(25, 40), topDiams = c(0,.1),
+                          startSeed = 6220)
R> sapply(sTrees3@trees, function(x) c(solidType = x@solidType, dbh = x@dbh,
+                                       topDiam = x@topDiam))
```

```
          tree.1 tree.2 tree.3 tree.4 tree.5 tree.6
solidType 9.7000 7.7000 9.1000 7.0000 8.4000 6.1000
dbh       0.2560 0.3200 0.3600 0.3144 0.3368 0.3435
topDiam   0.0055 0.0032 0.0011 0.0048 0.0244 0.0310
```

The Monte Carlo methods can now be applied to every sample (grid) point falling within each tree's inclusion zone. At each point, an estimate of the volume is derived using the appropriate method; the result is that even if we draw only one subsample for volume estimation at each grid point, when taken over the totality of grid cells within the inclusion zone, it can result in hundreds of subsamples for each tree (this, of course, depends on the grid resolution and the size of the tree's inclusion zone). For example, if a tree's inclusion zone contained 500 grid cell centers, it is like applying the methods described above on 500 subsampled heights and thus cross-sectional areas within the tree for volume estimation.

When using the two-stage approach in `sampSurf`, all arguments that are meant to be communicated to the "MonteCarloSampling" subclass object constructors are transferred through the dotted arguments (`...`) in a method call. This can happen in one of two places, depending on how one is constructing the objects, either the "long" or "short" way (e.g., Gove, 2012b).

Long: With this method one creates all the requisite intermediate objects individually (i.e., "Stem", "ArealSampling", "InclusionZone") during the course of creating a final sampling surface. All arguments that will be transferred to the subsampling methods *must* be specified when creating the "InclusionZone" subclass objects (see the examples for horizontal point sampling

in the following sections). All new additions to `sampSurf` that will support Monte Carlo subsampling will be designed this way.

Short: The arguments should be passed right in the `sampSurf` constructor. These will be passed down to the "InclusionZone" subclass constructor.

Note that the 'short' method may be less than satisfactory if more control is desired over the random number stream. This is because argument `startSeed` is used to set the random number stream in both the creation of "Stem" subclass objects, and in the Monte Carlo sampling methods. Therefore, when using the simple `sampSurf` constructor without intermediate objects, if `startSeed != NA`, the stream will be set to the same starting value for "Stem" subclass object creation, and again to initialize the Monte Carlo sampling. If `startSeed = NA`, then it does not matter.[7] This may not matter in many cases as the runs will still be replicable, it just depends on how fine a control is required by the user in creating objects.

12.1 The "MonteCarloSamplingIZ" Class

This is a virtual class specification that provides a link between the Monte Carlo methods described for individual stem objects and areal sampling methods through the "MonteCarloSampling" component described above.[8] It is defined as follows...

```
R> getClass('MonteCarloSamplingIZ')
```

```
Virtual Class "MonteCarloSamplingIZ" [package "sampSurf"]

Slots:

Name:              mcsObj         antithetic              proxy
Class: MonteCarloSampling           logical          character

Known Subclasses: "horizontalPointCMCIZ", "horizontalPointISIZ", "horizontalPointCVIZ"
```

12.1.1 Class slots

- *mcsObj*: An object that is a subclass of "MonteCarloSampling".

[7] Please see `?initRandomSeed` for details.

[8] This section may be skipped for those not interested in programming details.

- *antithetic*: `TRUE`: when sampling is conducted at each grid cell, the antithetic sampling variant to the method specified in the `mcsObj` slot will be applied. `FALSE`: the Monte Carlo method specified in the `mcsObj` slot will be applied.
- *proxy*: The name (character) of the proxy function used in the Monte Carlo sampling.

The system is designed so that extensions to areal sampling methods that employ any of the Monte Carlo methods described here must use this class to store the information about the subsampling method. It should be invoked via inheritance in a `contains` statement within the definition of the new areal class to make its slots available to the new class. The two-stage methods described in the following sections illustrate the concept (use `getClass` on these object names to see this).

The following presents some detail for class designers corresponding to what was discussed in §12 above. It is extremely important to note that for current and future such two-stage designs within `sampSurf`, the "InclusionZone" constructor for the method will create the above slots for the object. However, it must be recognized that the estimates in the `mcsObj` slot, while valid for the stem, are *not* the final estimate used in `sampSurf` for the stem. The information there is valid, but it is used only to pass desired options to the associated `izGrid` method for the class, where the actual areal estimates for the stem are generated for each grid cell within its inclusion zone. For example, if one wants to sample the current stem between height/length bounds that are a subset of the stem's total height/length, then the `segBnds` argument should be passed in the "InclusionZone" constructor for the two-stage object; this information is stored in the `mcsObj` slot above, and then used in the construction of the grid cell estimates in `izGrid`. Similarly, the `antithetic` slot must also be set in the "InclusionZone" class constructor via an appropriate argument to tell the associated `izGrid` method whether to apply antithetic sampling to the Monte Carlo estimates. As a final example, we also assign the `n.s` slot when the inclusion zone is created, not when the inclusion zone grid is determined. This has two effects: it restricts the "InclusionZoneGrid" object to just one subsample size (all grid cells have the same number of Monte Carlo subsample points for a given stem), but it also allows one to mix subsample sizes for different stems (i.e., one could have one subsample, another six).

The "InclusionZone" subclass constructors illustrated in the following sections use the above paradigm and their routines can be consulted for examples. For other details on the following methods, including class definitions, please see the corresponding help pages, as well as Gove (2012a) and Gove (2011b).

12.2 Two-stage horizontal point sampling

In this section, horizontal point sampling (HPS) is used to determine a tree's inclusion zone, and associated expansion factors. The following angle gauge will be used to determine the inclusion zone specifications...

```
R> ( aGauge = angleGauge(5) )


Object of class: angleGauge
------------------------------------------------------------
angle gauge method
------------------------------------------------------------

ArealSampling...
  units of measurement:  metric

angleGauge...
  Angle (ν) in degrees = 2.5625587 (153.75352 minutes)
  Angle (ν) in radians = 0.044725087
  Angle diopters (Δ) = 4.4754933
  Gauge constant (k) = 0.04472136
  Plot radius factor (prf) = 0.2236068 meters per cm (22.36068 meters per meter)
  Plot proportionality factor (α) = 44.7 meters per meter
  --Points...
    Basal area factor (baf) = 5 square meters per hectare
  --Lines...
    Diameter factor (df) = 134.16408 cm per hectare for a line segement of 20 meters
    Diameter factor (DF) = 11.18034 m per hectare for a line segement of 20 meters
```

Please note in what follows that, just as when the Monte Carlo methods described here are applied to individual trees, the attribute to be estimated is volume. The `sampSurf` constructor will allow you to ask for other attributes, but the result will be a surface with zero estimates for the stems in all inclusion zones. If you want to estimate other attributes like stem density under HPS, use the `horizontalPointIZ` constructor instead.

12.2.1 Crude Monte Carlo

In this example crude Monte Carlo sampling is used to subsample each tree at every sampling (grid) point within that tree's inclusion zone to estimate the tree's volume. First, the "horizontalPointCMCIZ" inclusion zones are determined for the collection of trees, then the sampling surface is developed from this latter collection of inclusion zones as usual. More detailed information on the "horizontalPointCMCIZ" class structure and construction of objects may be found in Gove (2012a). Note, however, that any argument in the `crudeMonteCarlo` constructor (§4) can be passed when creating objects of class "horizontalPointCMCIZ".[9]

[9] Indeed, construction of the inclusion zone object for this class is the only place one can specify such arguments and have them be used.

```
R> sT3.hpscmc.izs = standingTreeIZs(sTrees3, horizontalPointCMCIZ,
+                                   angleGauge = aGauge, startSeed = 989)
R> sT3.hpscmc.ss = sampSurf(sT3.hpscmc.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpscmc.ss)


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

Inclusion zone objects: horizontalPointCMCIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.3928074
  bias = 0.033061744
  bias percent = 0.98405495
  sum = 16624.756
  var = 75.755838
  st. dev. = 8.703783
  cv % = 256.53631
  surface max = 61.094665
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

The results of the sampling surface summary show that the method is unbiased, though it is possible to register a few percent bias using CMC at this grid resolution; however, the bias will be reduced as the grid resolution increases (cell size decreases), or the BAF is reduced, increasing the sample

for each tree[10]. Crude Monte Carlo is asymptotically unbiased; therefore, it may be somewhat the exception to register a small (less than one percent) bias on as few subsamples per tree as are shown above (on average 666.3 grid cells per tree). The surface is displayed in Figure 5. Note the lack of any discernible shape to the surface within inclusion zones: each time CMC is applied, a different (though always asymptotically unbiased) realization will result (unless sampling is constrained by the random number start seed).

```
R> plot(sT3.hpscmc.ss, useImage = FALSE)
```

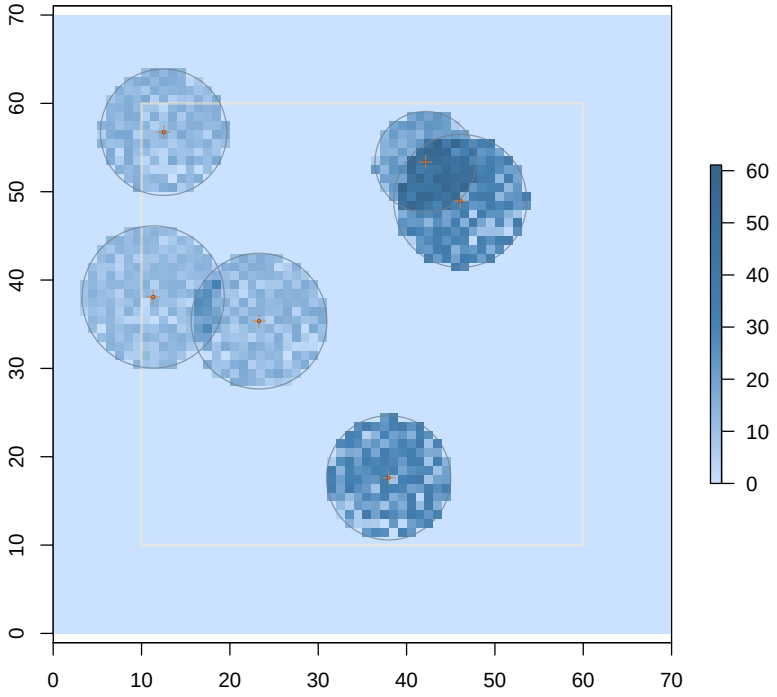

Figure 5: Sampling surface for the application of CMC within trees on a horizontal point sample inventory.

Just as the "horizontalPointCMCIZ" object construction is where one passes arguments to the CMC constructor in order to differentiate a sampling surface run, it is also where one specifies whether the antithetic variant is desired or not. The following results may be compared with the above to see that even with `n.s = 1` (the default), antithetic sampling improves on basic CMC...

[10]This can also be accomplished by using `n.s`> 1 for each tree as in antithetic CMC sampling.

```
R> sT3.hpsacmc.izs = standingTreeIZs(sTrees3, horizontalPointCMCIZ,
+                                    angleGauge = aGauge, antithetic = TRUE,
+                                    startSeed = 989)
R> sT3.hpsacmc.ss = sampSurf(sT3.hpsacmc.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpsacmc.ss)


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

Inclusion zone objects: horizontalPointCMCIZ (antithetic)
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.34679
  bias = -0.012955703
  bias percent = -0.38561558
  sum = 16399.271
  var = 65.781426
  st. dev. = 8.1105749
  cv % = 242.33893
  surface max = 46.771941
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

As expected, the standard deviation for antithetic is smaller (8.111) than that for straight CMC (8.704). This is corroborated graphically in Figure 6, which shows that the surface is generally less variable under antithetic CMC sampling.

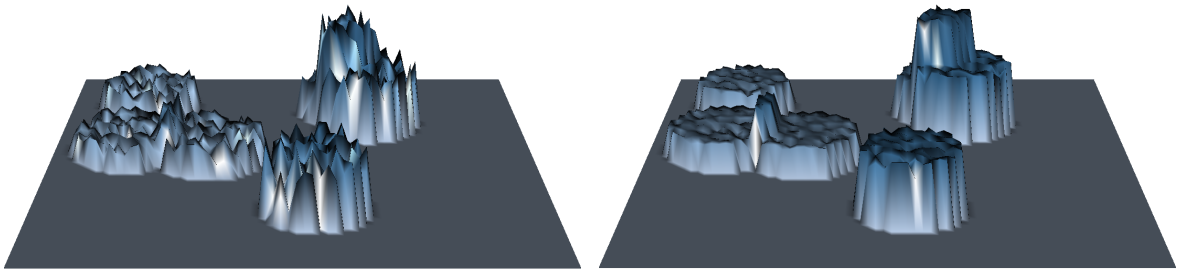

Figure 6: Sampling surfaces for CMC (left) and antithetic CMC (right).

12.2.2 Importance sampling

Importance sampling can be applied at each grid cell again by subsampling within a stem. Two examples are presented, the first uses the default ('`gvProxy`') proxy...

```
R> sT3.hpsis.izs = standingTreeIZs(sTrees3, horizontalPointISIZ,
+                                  angleGauge = aGauge, startSeed = 989)
R> sT3.hpsis.ss = sampSurf(sT3.hpsis.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpsis.ss)


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

Inclusion zone objects: horizontalPointISIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
```

```
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.4369811
  bias = 0.077235369
  bias percent = 2.2988457
  sum = 16841.207
  var = 101.55134
  st. dev. = 10.077268
  cv % = 293.20116
  surface max = 311.15401
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

Note that it is possible with the default '`gvProxy`', to generate proxy "cross sectional" areas that are very small on trees that do *not* taper to the tip. This will result in spikes in the sampling surface (abnormally large volume estimates). This can happen when $h_s \to H$ for some point s near the tip and consequently $g_s = H - h_s \to 0$, but the associated cross-sectional area, ρ_s, is still large since the tree has a broken top (i.e, the numerator in (5) stays large, while the denominator goes to zero). Thus, the estimated volume will inflate for this point if $n_s = 1$, or could simply force a larger average for the grid point if $n_s > 1$. When comparing the above results to those of CMC we can see that the maximum surface height of 311 is too large when compared to the CMC result of 61.1. In addition, comparing the surface variances it can be seen that the importance sampling variance is larger than the CMC variance. Both these are diagnostics that the above has occurred on at least one of the trees. Finally, note that the spikes in the estimates, though not numerous, were of sufficient magnitude to register a bias in the estimate. It can be shown through other examples that smaller spikes can still affect the variance without showing any noticeable bias.

The alternative proxy, '`wbProxy`', for the default taper function circumvents this as described in §8.3.3...

```
R> sT3.hpsis2.izs = standingTreeIZs(sTrees3, horizontalPointISIZ,
+                                   angleGauge = aGauge, proxy = 'wbProxy',
+                                   solidTypeProxy = 0.9, startSeed = 989)
R> sT3.hpsis2.ss = sampSurf(sT3.hpsis2.izs, smbtr)
```

```
Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpsis2.ss)


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

Inclusion zone objects: horizontalPointISIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.3371793
  bias = -0.022566365
  bias percent = -0.67166885
  sum = 16352.179
  var = 64.933603
  st. dev. = 8.0581389
  cv % = 241.46556
  surface max = 46.445363
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

Now the variance is less than the CMC variance, and the surface maximum of 46.4 is also less, which is more in line with theory. Note that antithetic CMC is as good as IS with this particular set of proxy parameters for IS. Of course, antithetic CMC requires two measurements per stem. The two IS surfaces can be contrasted in Figure 7.

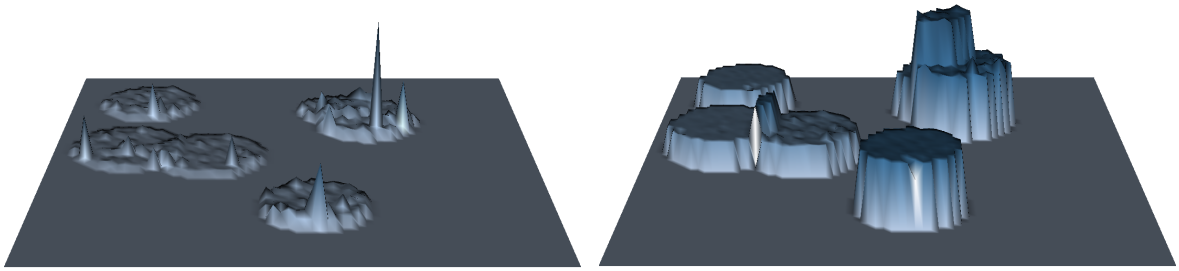

Figure 7: Sampling surfaces for IS with '`gvProxy`' (left) showing several "spikes" caused by inflated estimates. The results from the '`wbProxy`' (right) show a much more well-behaved (flatter) surface. Note that the vertical axis is different for the two surfaces (compare the surface maxima).

12.2.3 Control variate sampling

As with CMC and IS, CV sampling can be applied as a subsampling method at each grid cell within the HPS inclusion zone. It has already been pointed out that the '`gvProxy`' can generate a "biased" estimate (i.e., one with large error, §6.2), so we concentrate on using the '`wbProxy`' for `sampSurf`'s default taper model.

The first two examples use the default `truncateProxyStem = TRUE` so that any for tree that tapers to the tip, the associated proxy stem will be enlarged to approximately one cm top diameter (§8.3.3). The first example uses a proxy shape parameter that is close to the true taper shape for each tree...

```
R> sT3.hpscv.izs = standingTreeIZs(sTrees3, 'horizontalPointCVIZ',
+                                  angleGauge = aGauge, proxy = 'wbProxy',
+                                  solidTypeProxy = 0.9, startSeed = 989)
R> sT3.hpscv.ss = sampSurf(sT3.hpscv.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpscv.ss)
```

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

Inclusion zone objects: horizontalPointCVIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.4002807
  bias = 0.040535028
  bias percent = 1.206491
  sum = 16661.376
  var = 67.524701
  st. dev. = 8.2173415
  cv % = 241.66656
  surface max = 45.741359
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

Note that a small bias is recorded here due to the truncation and the slight mismatch of the shape models. There are two compensating forces in the bias statistic that will make this figure appear smaller than it is in absolute value. First, since the `solidTypeProxy` is smaller than the actual `stem@solidType`, the δ_s will tend to be positive on trees that have broken tops, because the true stem taper envelops the proxy taper. However, on stems that have top diameters of zero, the proxy taper will get "truncated" and therefore will tend to lie outside the true stem along at least some section of the upper bole, causing negative δ_s. The two are confounded, and since not all stems are truncated in the above collection, the positive and negative biases will cancel over the surface, resulting in what looks like a better estimate (bias-wise) than it undoubtedly is.

In the second example, the exact taper equation is used for the proxy stem, so the bias is isolated to the truncation component only...

```
R> sT3.hpscv2.izs = standingTreeIZs(sTrees3, 'horizontalPointCVIZ',
```

```
+                                    angleGauge = aGauge, proxy = 'wbProxy',
+                                    solidTypeProxy = NA, startSeed = 989)
R> sT3.hpscv2.ss = sampSurf(sT3.hpscv2.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpscv2.ss)


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

Inclusion zone objects: horizontalPointCVIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.2899287
  bias = -0.069816988
  bias percent = -2.0780438
  sum = 16120.651
  var = 63.335486
  st. dev. = 7.9583595
  cv % = 241.90067
  surface max = 44.216375
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

The result here is a negative bias as predicted above, and is the result of only truncation causing mismatch in the proxy and true stem taper (the proxy will be larger than the true taper due to the enlarged top diameter), leading to $\delta_s \leq 0$.

In the next example, the stems that taper to zero are not truncated, but we use the sample proxy shape parameter, not the true taper; this isolates the bias from mismatch of the proxy and true tree taper shapes...

```
R> sT3.hpscv3.izs = standingTreeIZs(sTrees3, 'horizontalPointCVIZ',
+                                   angleGauge = aGauge, proxy = 'wbProxy',
+                                   solidTypeProxy = 0.9, startSeed = 989,
+                                   truncateProxyStem = FALSE)
R> sT3.hpscv3.ss = sampSurf(sT3.hpscv3.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpscv3.ss)


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

Inclusion zone objects: horizontalPointCVIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.4535752
  bias = 0.093829455
  bias percent = 2.7927547
  sum = 16922.518
  var = 69.416186
  st. dev. = 8.3316377
  cv % = 241.24675
  surface max = 46.565459
  total # grid cells = 4900
  grid cell resolution (x & y) = 1 meters
```

```
  # of background cells (zero) = 3998
  # of inclusion zone cells = 902
```

The bias is positive here because the true taper envelops the proxy taper leading to $\delta_s \geq 0$.

Finally, we eliminate all sources of mismatch between the two taper models. CV sampling should be perfect in this example (see §6.2), and what "bias" is left is the pure "simulation bias" that is inherent in the grid resolution for the finite grid cell size used...

```
R> sT3.hpscv4.izs = standingTreeIZs(sTrees3, 'horizontalPointCVIZ',
+                                   angleGauge = aGauge, proxy = 'wbProxy',
+                                   solidTypeProxy = NA, startSeed = 989,
+                                   truncateProxyStem = FALSE)
R> sT3.hpscv4.ss = sampSurf(sT3.hpscv4.izs, smbtr)


Number of trees in collection = 6
Heaping tree: 1,2,3,4,5,6,


R> summary(sT3.hpscv4.ss)


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

Inclusion zone objects: horizontalPointCVIZ
Measurement units = metric
Number of trees = 6
True tree volume = 3.3597457 cubic meters
True tree basal area = 0.4930807 square meters
True tree surface area = 48.027403 square meters
True tree biomass = NA
True tree carbon = NA

Estimate attribute: volume
Surface statistics...
  mean = 3.3405525
  bias = -0.019193245
  bias percent = -0.57127077
  sum = 16368.707
```

```
var = 65.052783
st. dev. = 8.0655306
cv % = 241.44301
surface max = 44.22292
total # grid cells = 4900
grid cell resolution (x & y) = 1 meters
# of background cells (zero) = 3998
# of inclusion zone cells = 902
```

12.3 Critical height sampling and related variants

Classical critical height sampling is a spatially structured crude Monte Carlo variant for subsampling under horizontal point sampling. In addition, extensions to CHS that employ concepts from importance and antithetic sampling have been developed (Lynch and Gove, 2013). Both CHS and these derivative methods are available for use within `sampSurf`. The inclusion zone class structure and constructors for these methods are described in Gove (2012a). The various methods are associated with classes: "criticalHeightIZ", "importanceCHSIZ", "antitheticICHSIZ", and "pairedAICHSIZ".[11]

12.4 Variance estimation

The two-stage designs can have both within-stem variance and sampling variance associated with the overall variance estimate. As noted before, this had been discussed by Gregoire et al. (1986) and Valentine et al. (1990). By simply regarding the magnitude of the variance reported in these studies, which is similar to that found in the example runs previously described here for the within-stem component, one can be fairly certain that the sampling variance component will dominate the within-stem component. Additionally, the latter can only be estimated if more than one measurement per tree is taken. At the present time, there is no facility to automatically estimate a two-stage variance within `sampSurf` for such designs. The "lapSurf" class[12] will be of some help in this regard. In addition, the two-stage HPS-based methods described here all record the variance estimates for individual cells (i.e., the within-stem component) for each stem's inclusion zone within the respective "InclusionZoneGrid" object. Therefore, the pieces are all available and the combined variance estimate could be accomplished with a moderate amount of coding (less so when "lapSurf" is complete).

[11] Vignettes with comparisons of these methods are available from the author on request.
[12] Still in development.

13 Summary

Monte Carlo sampling methods make it convenient to estimate volume integrals through various subsampling methods. The addition of the "MonteCarloSampling" and "AntitheticSampling" classes provides for a full range of Monte Carlo methods within `sampSurf`. The respective subclasses are designed to work on "Stem" subclass objects, providing a mechanism to do simulations apart from any areal design. Moreover, these methods can be combined with virtually any areal sampling design in a two-staged fashion for design-unbiased estimation of volume; with examples applied to HPS given herein.

14 Acknowledgments

Tim Gregoire, Andrew Robinson and Harry Valentine reviewed either portions or all of this vignette. Their helpful comments and encouragement is very much appreciated.

A Closures in R

It might be helpful either as a refresher, or something new, to go over a simple set of examples illustrating closures in R. We'll present this as a sort of quiz to begin with. Some degree of failure is almost guaranteed on a few of these (unless you cheat) if you are unfamiliar with the concept of *closures* and *lexical scoping*, which are used in R. Closures are an integral part of *functional* languages such as Lisp and Scheme (on which R is partly based (Ihaka and Gentleman, 1996)), or newer languages such as Haskell. However, the concept may be somewhat foreign to those of us who came to R through Fortran or other procedural languages. Other versions of the S language, notably S-Plus, use a different scoping rule. Please note that a more extensive write-up on the subject with more realistic uses for closures in R is available from the author on request. In addition, a more formal coverage of these concepts is found in Gentleman and Ihaka (2000), as well as both Chambers (2008, p. 125) and Gentleman (2008, p. 59).[13] If you are new to R and have no intention of programming a proxy, then please skip this for now as it could only serve to confuse, and that is not the intended purpose of this appendix.

The real reason for this Appendix is for those who want to write their own proxy functions rather than using the built-in proxies within `sampSurf`. Because it might be useful to utilize the closure capabilities in R, as I have done with '`wbProxy`', understanding them would be advantageous at this point. Please note that you do not have to use the closure mechanism explicitly in your proxy function, neither of the built-in proxies '`cmcProxy`' and '`gvProxy`' utilize lexical scoping explicitly

[13] There are also some good explanations and examples of these concepts on the web; e.g., http://darrenjw.wordpress.com/2011/11/23/lexical-scope-and-function-closures-in-r/.

(but of course it is used implicitly regardless). Below we will talk a little more about how it is used within '`wbProxy`'.

A.1 Developing 'lexical intuition'

The question below in each example, is: "what is the value of `a` when `f` is evaluated?" First, something simple

```
R> a = 10
R> f = function() print(a)
R> f()
```

```
[1] 10
```

Obviously, we all passed this one, with `a = 10`.

A progressively less intuitive set of examples follows, see if you understand why **R** is doing what it is doing...

```
R> g1 = function() {a=5; print(a)}
R> g1()
```

```
[1] 5
```

Simple enough, and...

```
R> g2 = function() { f() }
R> g2()
```

```
[1] 10
```

Hopefully everyone is good to here. However,...

```
R> g3 = function() {a=5; f()}
```

What is the value of `g3()` when run? If you said 5 you are incorrect...

```
R> g3()
```

```
[1] 10
```

Why? Because of lexical scoping. If **R** used dynamic scoping, or **S**'s scoping rule,[14] the correct answer would have been 5. But under lexical scoping the correct answer is 10. So now it's time for an explanation. **R** evidently has two types of functions: closures and primitives (e.g., see `?closure`). All user-defined functions fall into the former class. In simple terms, a closure has an argument list, a body and an *enclosing environment*. The argument list should be self-evident, the body has **R** language statements and often a set of name-value pairs (i.e., the local variables and their assigned values) residing within the *evaluation environment*. The enclosing environment is the environment where the function was created or defined.[15] In the functions defined above, each of their enclosing environments is the global environment (`.GlobalEnv`), i.e., the **R** workspace.

Now, under dynamic scoping, no matter where a function is called from, bindings for unknown variables are searched for back up the call chain.[16] In the case of `g3`, dynamic scoping would look in the environment where `f` was called (i.e., within `g3`'s evaluation environment), and assign the value 5 to `a` within the call to `f`. And here's where lexical scoping differs: under lexical scoping, the value of an unknown object is searched for first *within the environment where the function was created* as stated above. Because `f` was created in the `.GlobalEnv`, that is its enclosing environment, and that is where **R** looks for the value to associate with the unknown `a` in the call to `print(a)` within `f`'s body. Recall that `a = 10` in `.GlobalEnv`, hence, regardless of where we call `f` from, it will always print the value of `a` found in `.GlobalEnv`, even if we change that value (as long as it exists within its enclosing environment). Now, also perhaps unintuitively, if `a` did not exist within `.GlobalEnv`, **R** will print an error to the effect that `a` is not found—it does not matter where `f` is called from (e.g., from within `g3` where `a` clearly has a definition), the result will be an error.[17] Even in the case of `g2` above, one will still be tempted to think that **R** is looking up through the call chain from `f` → `g2` → `.GlobalEnv` to find the value of `a`. But this is incorrect. The answer one gets would be consistent with what is printed above, but the logic is wrong. The correct logic under lexical scoping is: `f` → `.GlobalEnv`, bypassing `g2`'s environment completely.

Now try the following little example on your own to drive the above ideas home...

```
R> rm(a)
R> g3()
```

[14] Look first in the environment where the call has taken place, then look to the global environment (Gentleman and Ihaka, 2000).

[15] See also R Development Core Team, §2.1.5.

[16] Binding refers to the association of a value with a variable.

[17] Perhaps this has happened to you before, `a` is defined in `g3`, so it is clearly "available," but **R** does not see it under its scoping rules!

R can't find `a`, even though it clearly looks like it is defined within `g3` *before* the call to `f`. Now we understand why.

A.2 Digging a little more deeply

The above is all well and good, but what does it have to do with proxy functions and `sampSurf`? Well, in the following examples, we will take a look at a slightly different twist on the above, by returning a function that was created within another function. You should now be able to reason out what is going on without much help. This is exactly what is happening in the format of the proxy functions discussed in §8. Specifically, in §8.2, we see that each proxy function returns a `list` object, one component of which is a function called `g`, that evaluates the actual mathematical proxy. Thus, the function `g` is defined within the proxy (e.g., '`gvProxy`' or '`wbProxy`', etc.); under lexical scoping the environment where the function was defined is its closure. Keep this in mind in the following.

First, a slight twist on the last example...

```
R> a = 10
R> h = function() {function() print(a)}
R> h1 = h()
R> h1()
```

```
[1] 10
```

In this example we see that R will find a value for `a`. It looks first within the returned function's closure (the evaluation environment of `h`), and when it does not find the value, it goes up the call stack as we would expect to find the binding in `.GlobalEnv`. The reason it works here and not in the examples in the previous section, is that `.GlobalEnv` is the end of the line for searching both here and above. Since the closure for `g3` was `.GlobalEnv`, R had no further to search in the case where we had deleted `a`. But here, the sequence `h1` → `h` → `.GlobalEnv` is available because `h` is `h1`'s closure; therefore, when the binding is not found in `h`, R can search back up as far as necessary in the call stack to find the binding for `a`, in this case it was found in `.GlobalEnv`.

We can now amend the thinking above in the case where a function's enclosing environment is not `.GlobalEnv` but that of another function. If a variable binding is not presented within the function itself (through direct assignment or an argument), then R will first look in the enclosing environment for the binding, and if it is not found there, R will employ a search back along the chain of successive enclosing environments for the binding of an unknown variable until it is found or it reaches `.GlobalEnv` (e.g., Chambers, 2008, p. 120).

Finally, here is an example using the above rule that would be directly applicable to the proxy function closure concept in `sampSurf`...

```
R> h2 = function() {a=2
+                   cat("\nh2's environment: ")
+                   print(environment())
+                   f2 = function() print(a)
+                   return(f2) }
R> ff = h2()

h2's environment: <environment: 0x7d2ea60>

R> ff()

[1] 2

R> environment(ff)

<environment: 0x7d2ea60>

R> ls(envir = environment(ff))

[1] "a"  "f2"
```

In this example we see that even though `f2` was called in the `.GlobalEnv`, through a function assigned (but not created/defined) there (i.e., `ff` through the return in the call to `h2`), it does not return the value `a = 10` in `.GlobalEnv`, but rather first searches `ff`'s (i.e., `f2`'s) enclosing environment for the variable binding. In this case, the value of `a` defined within the evaluation environment of `h2` (which is `f2`'s enclosing environment) is used, no further searching is necessary. Again, `f2`'s enclosing environment is the evaluation environment of `h2` and persists as long as required (e.g., until `ff` is deleted).

In the proxy function '`wbProxy`', as we have mentioned, lexical scoping and closures are explicitly utilized. Each proxy method is required to have the argument `stem` in its argument list (§8.2). This is a "Stem" subclass object from `sampSurf` corresponding to the "downLog" or "standingTree" object that we are subsampling within. We can therefore make changes to this object within '`wbProxy`' that will transform it into a proxy `stem` object. Then, when the `g` function is created

within '`wbProxy`' and returned in the `list` from a call to '`wbProxy`', `g`'s closure is the evaluation environment of '`wbProxy`' (so far so good); therefore, this modified (now proxy) `stem` that was created within '`wbProxy`' still exists and resides within `g`'s closure, no matter where it is called from. So we always have the proxy `stem` defined within '`wbProxy`' around as long as `g` is around, and `g` will always look first to its closure to find the binding for `stem`, so we are certain that the proxy stem will always be used. As stated in §8.3.4, this is why `g` will always find the proxy `stem` object to determine the proxy cross-sectional area correctly in the subsampling routines. Moreover, each call to '`wbProxy`' with a different `stem` argument (and thus a different "Stem" object) will create a new environment (closure) for that object, assuring that the returned `g` proxy applies to *that* particular instance of the "Stem" object. In this way, closures allow for a form of data encapsulation, much like in object-oriented programming (Gentleman and Ihaka, 2000).

That's probably enough. Again, I must stress that this is not difficult, and a little playing with similar examples should drive the concepts home. One final caveat, which I probably should not mention: **R** allows one to change the enclosing environment for a function. If the enclosing environment of a function is changed in the sense of assigning a whole new environment to it, then the results will change accordingly. One must do this explicitly within some **R** code—don't do it unless there is a very good reason to and until the above is clearly understood. More importantly, because you can change the closure environment, you can add variable bindings to the closure, or change variable values. This a more advanced subject, some examples of this are available on request from the author, or simply experiment with it yourself.

References

J. M. Chambers. *Software for Data analysis: Programing with R.* Springer, 2008. 20, 53, 56

R. Gentleman. *R Programming for Bioinformatics.* Chapman & Hall/CRC, Boca Raton, 2008. 20, 53

R. Gentleman and R. Ihaka. Lexical scoping and statistical computing. *Journal of Computational and Statistical Graphics*, 9(3):491–508, 2000. 53, 55, 58

J. H. Gove. Propagating probability distributions of stand variables using sequential Monte Carlo methods. *Forestry*, 82(4):403–418, 2009. 8

J. H. Gove. *The "InclusionZone" Class*, 2012a. URL `http://CRAN.R-project.org/package=sampSurf`. `sampSurf` package vignette. 39, 40, 52

J. H. Gove. *The "sampSurf" Class*, 2012b. URL `http://CRAN.R-project.org/package=sampSurf`. `sampSurf` package vignette. 37

J. H. Gove. *The "Stem" Class*, 2011a. URL `http://CRAN.R-project.org/package=sampSurf`. `sampSurf` package vignette. 2, 4, 7, 22

J. H. Gove. *The "InclusionZoneGrid" Class*, 2011b. URL http://CRAN.R-project.org/package=sampSurf. `sampSurf` package vignette. 39

T. G. Gregoire and H. T. Valentine. *Sampling strategies for natural resources and the environment*. Applied environmental statistics. Chapman & Hall/CRC, N.Y., 2008. 2, 5, 8, 10, 22

T. G. Gregoire, H. T. Valentine, and G. M. Furnival. Estimation of bole volume by importance sampling. *Canadian Journal of Forest Research*, 16:554–557, 1986. 36, 52

J. M. Hammersley and D. C Handscomb. *Monte Carlo methods*. Chapman and Hall, 1979. 10

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. 53

T. B. Lynch and J. H. Gove. An antithetic variate to facilitate upper-stem height measurements for critical height sampling with importance sampling. *Canadian Journal of Forest Research*, 2013. (In review). 52

R Development Core Team. *R language definition*. 55

R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo method*. John Wiley and Sons, 2nd edition edition, 2008. 8, 10

H. T. Valentine, T. G. Gregoire, and G. M. Furnival. Importance sampling for volume with a portable computer. In V. J. LaBau and T. Cunia, editors, *State-of-the-Art methodology of forest inventory: A symposium proceedings*, PNW-GTR-263, pages 88–95. U. S. Forest Service, Pacific Northwest Research Station, 1990. 36, 52