The `selfthin` scripts

Lars Hellström

2012–2017

Abstract

The following code performs various simple simulations of branch growth tempered by self-thinning. It is primarily a library of subroutines that can be combined to perform such simulations, but included are also a variety of scripts that demonstrate such combinations.

There are furthermore utilities for exporting the simulated results into a variety of formats: SQLite databases, CSV files, Matlab tree-lists, and in particular MetaPost graphics. Not all of the export options have been in use throughout the development process, so some of them may be incomplete.

Contents

Preamble

This document are Literate Programming sources in the DocStrip style (as used for most things LaTeX). To generate the stripped source files expected by an interpreter, one typically runs LaTeX on an accompanying `.ins` file, which copies selected modules from this file to a new file.

The main module is named pkg. In Sections 7 and 8 there are a number of modules (e.g. GUI20130122, simplerun20170510, branchgrowfig) that (together with pkg) produce scripts for specific runs. The database module is for storing data in a database (usually SQLite via TDBC interface); the alternative would be to output such data to CSV files.

1 Data structure

The main data structure is a standard data-tree, where nodes have one of the two formats

```
bud {} {}
node {attr-dict} {child-list}
```

The {*attr-dict*} is presently only used for fine details of data export (which nodes should be marked?). It is sort-of implied that there are metamers at the root ends of `node`s, but let's not worry about that at the moment.

2 Growth

The growth operation maps each `bud` to a `node` with μ `bud` children, and each `node` to itself (recursively applying the growth operation to the children, though). It is implemented as a data-is-code operation with call syntax

```
namespace inscope selfthin::grow {tree} {mu}
```

⟨*pkg⟩

```
namespace eval selfthin::grow {}
```

`selfthin::grow::bud` (proc)

The `bud` procedure thus has the call syntax

```
selfthin::grow::bud {attr} {dummy-children} {mu}
```

```
proc selfthin::grow::bud {attr children mu} {
   list node $attr [lrepeat $mu {bud {} {}}]
}
```

`selfthin::grow::node` (proc)

The `node` procedure similarly has the call syntax

```
node {attr} {children} {mu}
```

```
proc selfthin::grow::node {attr children mu} {
   set L {}
   foreach child $children {lappend L [{*}$child $mu]}
   return [list node $attr $L]
}
```

2.1 Random branching growth

The following variation on the basic growth operation instead randomly selects the number of `bud` children for each new `node`, treating μ as a stochastic variable. It is implemented as a data-is-code operation with call syntax

> `namespace inscope selfthin::rgrow` {*tree*} {*distribution*}

where {*distribution*} is a list of real numbers that give the probability distribution. The first element should be `-Inf`. Element k should be the probability that $\mu < k$, i.e., $\sum_{i=0}^{k-1} \mathrm{P}[\mu = i]$. Only probabilities < 1 need to be explicitly included in the list.

```
namespace eval selfthin::rgrow {}
```

`selfthin::rgrow::bud` (proc)

The `bud` procedure thus has the call syntax

> `bud` {*attr*} {*dummy-children*} {*distribution*}

```
proc selfthin::rgrow::bud {attr children distribution} {
   list node $attr [lrepeat [
      lsearch -real -sorted -bisect $distribution [expr {rand()}]
   ] {bud {} {}}]
}
```

`selfthin::rgrow::node` (proc)

The `node` procedure similarly has the call syntax

> `node` {*attr*} {*children*} {*distribution*}

```
proc selfthin::rgrow::node {attr children distribution} {
   set L {}
   foreach child $children {lappend L [{*}$child $distribution]}
   return [list node $attr $L]
}
```

2.2 Growth with identity

Another variation on the basic growth operation is to equip each `node` and `bud` with an identity, to allow tracing them as the tree ages. The call syntax is simply

> `namespace inscope selfthin::igrow` {*tree*} {*mu*}

which returns the grown {*tree*}. What happens is that each new `bud` gets an `id` attribute obtained by `lappend`ing its position index to the `id` attribute of the parent (that transforms into a `node`).

```
namespace eval selfthin::igrow {}
```

selfthin::igrow::bud (proc)

The bud procedure thus has the call syntax

bud {*attr*} {*dummy-children*} {*mu*}

This works by copying all attributes from parent to children, modifying only the id one (which is usually the only one there is).

```
proc selfthin::igrow::bud {attr children mu} {
   set children {}
   while {[llength $children] < $mu} {
      set cattr $attr
      dict lappend cattr id [llength $children]
      lappend children [list bud $cattr {}]
   }
   return [list node $attr $children]
}
```

selfthin::igrow::node (proc)

The node procedure similarly has the call syntax

node {*attr*} {*children*} {*mu*}

```
proc selfthin::igrow::node {attr children mu} {
   set L {}
   foreach child $children {lappend L [{*}$child $mu]}
   return [list node $attr $L]
}
```

3 Counting

The histogram operation returns a list of counts of items at different distances from the root in the tree. The call syntax is

namespace inscope selfthin::histogram {*tree*}

```
namespace eval selfthin::histogram {}
```

selfthin::histogram::bud (proc)

A bud has no children, so the return value is the list of 1.

```
proc selfthin::histogram::bud {attr children} {list 1}
```

selfthin::histogram::node (proc)

A node goes through its children, sums the lists up, and then prepends a 1 for itself.

```
proc selfthin::histogram::node {attr children} {
   if {![llength $children]} then {return [list 1]}
   set sum [{*}[lindex $children 0]]
   foreach child [lrange $children 1 end] {
      set term [{*}$child]
      while {[llength $term] < [llength $sum]} {lappend term 0}
      while {[llength $term] > [llength $sum]} {lappend sum 0}
      set newsum {}
      foreach a $sum b $term {lappend newsum [expr {$a+$b}]}
      set sum $newsum
   }
   return [list 1 {*}$sum]
}
```

The `harvest` operation returns a list of all subtrees at a given height. The call syntax is

`namespace inscope selfthin::harvest` {*tree*} {*height*}

where $height = 0$ corresponds to returning the (one-element list whose only element is the) {*tree*}.

```
namespace eval selfthin::harvest {}
```

`selfthin::harvest::node` (proc)

The main case is that of cutting at a node. This is most efficient at $height = 1$, but $height = 0$ must also be supported. For $height > 1$, there is a straightforward recursion.

```
proc selfthin::harvest::node {attr children height} {
   if {$height == 1} then {
      return $children
   } elseif {$height > 1} then {
      incr height -1
      set res {}
      foreach child $children {
         lappend res {*}[{*}$child $height]
      }
      return $res
   } else {
      return [list [list node $attr $children]]
   }
}
```

`selfthin::harvest::bud` (proc)

A `bud` has no children, so only the $height = 0$ case can return anything nonempty.

```
proc selfthin::harvest::bud {attr children height} {
   if {$height == 0} then {
      return [list {bud {} {}}]
   } else {
      return {}
   }
}
```

3.1 Work with identities

The `identities` operation returns the flat list of all `id`s found in a tree. The call syntax is

`selfthin::identities` {*tree*}

Since this treats all node types equally, it does not need to be a data-is-code operation.

`selfthin::identities` (proc)

It is however highly recursive.

```
proc selfthin::identities {tree} {
   set res {}
   foreach child [lindex $tree 2] {
      lappend res {*}[identities $child]
   }
   if {[dict exists [lindex $tree 1] id]} then {
      lappend res [dict get [lindex $tree 1] id]
   }
   return $res
}
```

`selfthin::aritybyid` (proc)

The `aritybyid` procedure is similar, but returns a dictionary mapping `id` to arity of the node is question. The idea is that this allows for restricting to branching points.

```
proc selfthin::aritybyid {tree} {
   set call [list ::dict merge]
   foreach child [lindex $tree 2] {
      lappend call [aritybyid $child]
   }
   if {[dict exists [lindex $tree 1] id]} then {
      lappend call [dict create [dict get [lindex $tree 1] id]\
                                        [llength [lindex $tree 2]]]
   }
   return [{*}$call]
}
```

`selfthin::markwhere` (proc)

This procedure sets an attribute on all `node`s and `bud`s in a tree where the value of an(other) attribute passes a custom test. The call syntax is

`selfthin::markwhere` {*tree*} {*select-attr*} {*select-prefix*} {*set-attr*} {*set-value*}

and the return value is a modified {*tree*}. The {*select-prefix*} is a command prefix with the syntax

⟨*select-prefix*⟩ {*value*}

that returns a boolean, where {*value*} is the value of the {*select-attr*} in the attribute dictionary of a node. If the ⟨*select-prefix*⟩ call returns true, then node attributes are modified by setting the {*set-attr*} attribute to {*set-value*}.

To set up a set membership test, encode the set as a dictionary and use `dict\ exists` {*set*} as ⟨*select-prefix*⟩.

This, too, is highly recursive.

```
proc selfthin::markwhere {tree selname selprefix setname setval} {
   foreach {type attrD children} $tree {break}
   if {[dict exists $attrD $selname] && [
      {*}$selprefix [dict get $attrD $selname]
   ]} then {
      dict set attrD $setname $setval
   }
   return [list $type $attrD [lmap child $children {
      markwhere $child $selname $selprefix $setname $setval
   }]]
}
```

4 Thinning

4.1 Deterministic thinning

The `up1d` operation performs thinning of a tree by thinning subtrees that exceed a size bound. The thinning is performed by dropping a branch, not one at the node in which the size bound violation is detected, but at the next branching above—hence the `up1`. The `d` is because the thinning is deterministic rather than pseudorandom, although the rule (as explained below) is somewhat arbitrary.

The call syntax is

`namespace inscope selfthin::up1d` {*tree*} {*limits-list*}
{*limits-index*} {*seed*}

and the return value is a list

{*thinned tree*} {*bud-counts-list*}

As a transformation on the {*tree*}, the result is the {*thinned tree*}. The {*bud-counts-list*} is a list, with one element for each child of the first `node` that is a branching point, of how many `bud`s the respective child tree contains. If there is no branching anywhere, then the {*bud-counts-list*} has one element.

The {*limits-list*} is a list of limits on the number of buds a subtree can have without being subjected to thinning. The {*limits-index*} is the index of the position in the list that is relevant for the root of the {*tree*}; this is decremented by 1 in each recursive call. No thinning happens if the {*limits-index*} is negative. The {*seed*} is an integer which is subjected to modulo operations to choose which subbranch to prune; the same {*seed*} value is used in recursive calls.

```
namespace eval selfthin::up1d {}
```

`selfthin::up1d::bud` (proc)

Buds have no branching, and are one bud.

```
proc selfthin::up1d::bud {attr children limitsL limitsIdx seed} {
   list {bud {} {}} [list 1]
}
```

`selfthin::up1d::node` (proc)

For a `node`, the first order of business is to recursively apply the operation to all children.

```
proc selfthin::up1d::node {attr children limitsL limitsidx seed} {
   set childL {}
   set coundsL {}
```

```
     set nextidx [expr {$limitsidx-1}]
     foreach child $children {
        set res [{*}$child $limitsL $nextidx $seed]
        lappend childL [lindex $res 0]
        lappend countsL [lindex $res 1]
     }
```
Next check if there is pruning to do. The check (and thus the pruning) is skipped if there is only one child, since in that case something in that child should have already checked a tighter limit.
```
     if {[llength $childL] > 1 && $limitsidx>=0 &&\
                              [::tcl::mathop::+ {*}[concat {*}$countsL]] >\
                                       [lindex $limitsL $limitsidx]} then {
```
i is the index of the child that will be subjected to pruning. If there is no branch point in that subtree, then the whole child is dropped.
```
        set i [expr {$seed % [llength $childL]}]
        if {[llength [lindex $countsL $i]] <= 1} then {
           set childL [lreplace $childL $i $i]
           set countsL [lreplace $countsL $i $i]
        } else {
```
Otherwise one branch (with index j) of the child is pruned. This branch may sit several levels down in the recursion, so `pruneIL` is constructed to hold the list of indices that will address the right list of children.
```
           set pruneIL [list $i 2]
           set subchildL [lindex $childL $pruneIL]
           while {[llength $subchildL] == 1} {
              lappend pruneIL 0 2
              set subchildL [lindex $subchildL 0 2]
           }
           set j\
                [expr {($seed / [llength $childL]) % [llength $subchildL]}]
           lset childL $pruneIL [lreplace $subchildL $j $j]
           lset countsL $i [lreplace [lindex $countsL $i] $j $j]
        }
     }
```
Now we are ready to return the result. If there is only one child left, then use the list of counts for that. Otherwise add up the count lists for each child.
```
     if {[llength $childL] == 1} then {
        set newcounts [lindex $countsL 0]
     } else {
        set newcounts {}
        foreach L $countsL {
           lappend newcounts [::tcl::mathop::+ {*}$L]
        }
     }
```
And return the result.
```
     return [list [list node $attr $childL] $newcounts]
  }
```

4.2 Mildly stochastic thinning

The `up1r` operation is similar to the `up1d` operation in that it performs thinning of a tree by thinning subtrees that exceed a size bound, but it adds some randomness to the process, to give it more the appearance of a simulation.

The thinning is again performed by dropping a branch, not one at the node in which the size bound violation is detected, but at the next branching above—hence the `up1`. The `r` is of course for being random, although the randomness source (the `rand` function) is not of particularly high quality. There are two things that are subject to randomness:

- The choice of which branch to drop is random. The algorithm is a simple multiply uniform random $]0, 1[$ by number-of-choices and truncate to produce target index.
- There is also a certain probability p for ignoring a size bound violation. Linus suggested this, and it is extremely easy to implement, so why not?

The recommended calling syntax is

`selfthin::up1r` {*tree*} {*limits-list*} {*limits-index*} {*ignore-probability*}

which returns a self-thinned form of the {*tree*}. The {*limits-list*} is the list of those sizes which will trigger dropping of some subbranch when exceeded, and the {*limits-index*} is the index into this list which applies for the root of the {*tree*}; this will then be decremented by 1 for each `node` the operation recurses into. The {*ignore-probability*} is the probability ($p \in [0, 1]$) that no subbranch is dropped even after a size violation has been detected, at one node, and this time the call is made.

There is also a more bare data-is-code operation, for which the call syntax is

`namespace inscope selfthin::up1r` {*tree*} {*limits-list*}
{*limits-index*} {*ignore-probability*}

and the return value is a pair

{*thinned tree*} {*bud-counts-list*}

As a transformation on the {*tree*}, the result is the {*thinned tree*}. The {*bud-counts-list*} is a list, with one element for each child of the first `node` that is a branching point, of how many `bud`s the respective child tree contains. If there is no branching anywhere, then the {*bud-counts-list*} has one element.

```
namespace eval selfthin::up1r {}
```

`selfthin::up1r::bud` (proc)

Buds have no branching, and are one bud.

```
proc selfthin::up1r::bud {attr children limitsL limitsIdx p} {
   list {bud {} {}} [list 1]
}
```

`selfthin::up1r::node` (proc)

For a `node`, the first order of business is to recursively apply the operation to all children.

```
  proc selfthin::up1r::node {attr children limitsL limitsidx p} {
     set childL {}
     set coundsL {}
     set nextidx [expr {$limitsidx-1}]
     foreach child $children {
        set res [{*}$child $limitsL $nextidx $p]
        lappend childL [lindex $res 0]
        lappend countsL [lindex $res 1]
     }
```

Next check if there is pruning to do. The check (and thus the pruning) is skipped if there is only one child, since in that case something in that child should have already checked a tighter limit.

```
     if {[llength $childL] > 1 && $limitsidx>=0 &&\
                             [::tcl::mathop::+ {*}[concat {*}$countsL]] >\
                        [lindex $limitsL $limitsidx] && rand()>=$p} then {
```

i is the index of the child that will be subjected to pruning. If there is no branch point in that subtree, then the whole child is dropped.

```
        set i [expr {int(rand()*[llength $childL])}]
        if {[llength [lindex $countsL $i]] <= 1} then {
           set childL [lreplace $childL $i $i]
           set countsL [lreplace $countsL $i $i]
        } else {
```

Otherwise one branch (with index *j*) of the child is pruned. This branch may sit several levels down in the recursion, so `pruneIL` is constructed to hold the list of indices that will address the right list of children.

```
           set pruneIL [list $i 2]
           set subchildL [lindex $childL $pruneIL]
           while {[llength $subchildL] == 1} {
              lappend pruneIL 0 2
              set subchildL [lindex $subchildL 0 2]
           }
           set j [expr {int(rand()*[llength $subchildL])}]
           lset childL $pruneIL [lreplace $subchildL $j $j]
           lset countsL $i [lreplace [lindex $countsL $i] $j $j]
        }
     }
```

Now we are ready to return the result. If there is only one child left, then use the list of counts for that. Otherwise add up the count lists for each child.

```
     if {[llength $childL] == 1} then {
        set newcounts [lindex $countsL 0]
     } else {
        set newcounts {}
        foreach L $countsL {
           lappend newcounts [::tcl::mathop::+ {*}$L]
```

```
      }
   }
```

And return the result.

```
   return [list [list node $attr $childL] $newcounts]
}
```

selfthin::up1r (proc) The top-level up1r procedure has as one task to strip off the {*bud-counts-list*} returned by the bare operation, but it also performs an extra self-thinning step just like node, to ensure that every node may lose a branch.

```
proc selfthin::up1r {tree limitsL limitsidx p} {
   foreach {tree counts} [
      namespace inscope up1r $tree $limitsL $limitsidx $p
   ] break
```

Next check if there is pruning to do. The check for having multiple children is dropped, or perhaps rather replaced with a check that there is a branch point somewhere in the tree.

```
   if {$limitsidx>=0 && rand()>=$p && [llength $counts]>1 &&\
      [::tcl::mathop::+ {*}$counts] > [lindex $limitsL $limitsidx]} then\
                                                                       {
```

Otherwise one branch (with index j) of the child is pruned. This branch may sit several levels down in the recursion, so pruneIL is constructed to hold the list of indices that will address the right list of children.

```
      set pruneIL [list 2]
      set subchildL [lindex $tree $pruneIL]
      while {[llength $subchildL] == 1} {
         lappend pruneIL 0 2
         set subchildL [lindex $subchildL 0 2]
      }
      set j [expr {int(rand()*[llength $subchildL])}]
      lset tree $pruneIL [lreplace $subchildL $j $j]
   }
   return $tree
}
```

Simulations indicate that the root rule applied by up1r does not match the general node rule of up1r::node—the root rule leads on average to a tree whose first branch point comes quite early (within the first four levels on a tree of height 100), followed by a sharp decrease in the rate at which $g(k,n)$ grows with k. On retrospect, this should not be that surprising; the above root rule only comes into play in cases where the first branch point would otherwise remain oversaturated after thinning, and that is way less often than one might see pruning at other branch points. Hence there could be a point in seeking a root rule that harmonises better with the general node rule.

One approach to devising such an alternative root rule is to emulate a virtual branch point below the root, and apply the ordinary node rule logic (even if not its implementation) to that branch point. The size of the virtual branch meeting the full tree can simply be taken to be equal to that of the tree, and this far down in the tree it would be unusual with a branching of degree higher than 2, even if the underlying μ is larger. The tricky matter is however how far down below the root that the virtual branch point should be considered to be, as that determines the bound against which the tree size should be compared.

`selfthin::up1r_Poisson` (proc)

The `up1r_Poisson` procedure answers the question about the position of the virtual branching point by making it (the distance from root to virtual branching point) a Poisson distributed random variable. The argument for a Poisson distribution is mainly that it "has the right shape"—one may prescribe the average, there is a maximum around that average, and there is no arbitrary upper cut-off—but one can also make a slightly more analytical argument for why it should be about right. First, the Poisson distribution arises as the $n \to \infty$ limit of the binomial distribution when one keeps the expected value fixed. Second, the binomial distribution is a sum of a number of independent identically distributed $\{0, 1\}$ variables. Third, the length of the path between two branching points does indeed reflect a number of times that 1 was chosen rather than 0, namely the number of self-thinnings which went down that branch and not its sibling (although the true distribution there is probably much more complicated than a simple binomial distribution).

The call syntax for this procedure is

`up1r_Poisson` {*tree*} {*limits-list*} {*limits-index*} {*ignore-probability*} {*dimensionality*}

The first four arguments are as for `up1r`. The fifth {*dimensionality*} argument is/should be the exponent in the asymptotic growth of the {*limits-list*} elements. It is used for estimating the expected position of the virtual branching point, as explained below.

As with `up1r`, the first step is the self-thinning of the tree based on real nodes in it.

```
proc selfthin::up1r_Poisson {tree limitsL limitsidx p d} {
   foreach {tree counts} [
      namespace inscope up1r $tree $limitsL $limitsidx $p
   ] break
```

The second step is to check whether there is some pruning to do. Unlike the case in a normal node, the choice of whether to go down this branch (as opposed to its virtual sibling) is done first, because it is much less work than checking whether the saturation bound has been reached. Also, it can be combined with the {*ignore-probability*} randomness for not pruning even if saturation is exceeded.

```
   if {[llength $counts]<=1 || 2*rand()<1+$p} then {return $tree}
```

Next, the expected distance λ down to the virtual branching point is computed; this will then determine the distribution of the random variable. The idea motivating this calculation is that the distance between branching points decreases by a factor $2^{1/d}$ for every branching point one passes (because that is what it takes to stay on the saturation limit), so the length of the root segment can be approximated by taking an average of lengths higher up in the tree, appropriately scaled. Arbitrarily, the average is taken with a recursion depth of two.

```
      set L [up1r_Poisson,sample $tree 2 1 [expr {pow(2,1.0/$d)}]]
      set lambda [expr {[
         ::tcl::mathop::+ {*}[lrange $L 1 end]
      ]/([llength $L]-1) - [lindex $L 0]}]
```

Then comes the sampling of the Poisson random variable. The algorithm used for this is the one of Knuth, which roughly amounts to adding exponentially distributed continuous random variables until the sum exceeds the sought λ, and then declaring the number of terms the sampled value of the Poisson random variable. Although instead of stepping a separate variable, `limitsidx` is increased by one for each term.

```
      set E [expr {exp(-$lambda)}]
      set prod 1.0
      while {$prod > $E} {
         incr limitsidx
         if {$limitsidx >= [llength $limitsL]} then {return $tree}
         set prod [expr {$prod*rand()}]
      }
```

So, now there is *finally* a pruning check to carry out.

```
      if {2*[::tcl::mathop::+ {*}$counts] > [lindex $limitsL $limitsidx]}\
                                                                   then {
         set pruneIL [list 2]
         set subchildL [lindex $tree $pruneIL]
         while {[llength $subchildL] == 1} {
            lappend pruneIL 0 2
            set subchildL [lindex $subchildL 0 2]
         }
         set j [expr {int(rand()*[llength $subchildL])}]
         lset tree $pruneIL [lreplace $subchildL $j $j]
      }
      return $tree
   }
```

selfthin::up1r_Poisson,sample (proc)

This is a helper procedure for `up1r_Poisson` which samples the distances between branching points in a tree. The call syntax is

`up1r_Poisson,sample` {*tree*} {*depth*} {*weight*} {*factor*}

where {*tree*} is the tree to sample and {*depth*} is how deep the recusion should be. If depth is 0, then the data returned reflects only the distance to the first branching point in the {*tree*}, but for every 1 that the {*depth*} is increased, another layer of branching points are sampled.

The return value is a list of sample values, with the first one being for the first (rootmost) branching point. All sample values are weighted; those at this level by a factor {*weight*}, and for each subsequent level the weight is multiplied by the {*factor*}.

```
   proc selfthin::up1r_Poisson,sample {tree depth weight factor} {
```

```
   set n 1
   while {[llength [lindex $tree 2]] == 1} {
      incr n
      set tree [lindex $tree 2 0]
   }
   set res [list [expr {$n * $weight}]]
   if {[incr depth -1] < 0} then {return $res}
   set weight [expr {$weight*$factor}]
   foreach child [lindex $tree 2] {
      lappend res {*}[
         up1r_Poisson,sample $child $depth $weight $factor
      ]
   }
   return $res
}
```

4.3 Non-uniform stochastic thinning

The point of the operation is to take stress into account when picking branches to prune: the more in excess of carrying capacity a branch is, the higher will the probability be of a pruning operation picking just that (among its siblings and cousins). Define the stress level x of a branch to be the quotient of number of buds to carrying capacity. If $x_1, \dots, x_n$ are the stress levels of the branches that a pruning candidates, then the probability of picking branch i should be

$$\frac{x_i^q}{\sum_{j=1}^n x_j^q} \tag{1}$$

where the exponent q is a parameter of the model. $q = 0$ reproduces uniform probability (except that it's one-step uniform, rather than two uniform steps as for `up1d` and `up1r`; there is a difference only where branching arity is not constant, which requires $\mu > 2$). Higher values of q concentrates the weight to stressed branches.

In order to make these calculations convenient, the bare `up1n` operation needs to return a bit more information about the thinned tree than the previous operations. The return value is a list

{*tree*} {*subbranch-statuses*} {*self-status*}

where a *status* is a pair

{*bud-count*} {*bud-limit*}

The call syntax is

`namespace inscope selfthin::up1n` {*tree*} {*limits-list*}
    {*limits-index*} {*ignore-probability*} {*weighting-exponent*}

As before, there is also a wrapped calling syntax

`selfthin::up1n` {*tree*} {*limits-list*} {*limits-index*} {*ignore-probability*} {*weighting-exponent*}

which returns a self-thinned form of the {*tree*}. This also applies a root rule, to allow pruning at the first branching point.

```
namespace eval selfthin::up1n {}
```

`selfthin::up1n::bud` (proc)

Buds have no branching, and are one bud.

```
proc selfthin::up1n::bud {attr children limitsL limitsIdx p q} {
   list [list bud $attr {}] {} [list 1 [lindex $limitsL $limitsIdx]]
}
```

`selfthin::up1n::node` (proc)

For a `node`, the first order of business is to recursively apply the operation to all children.

```
proc selfthin::up1n::node {attr children limitsL limitsidx p q} {
   set childL {}
   set substatusL {}
   set selfstatusL {}
   set buds 0
   set nextidx [expr {$limitsidx-1}]
   foreach child $children {
      set res [{*}$child $limitsL $nextidx $p $q]
      lappend childL [lindex $res 0]
      lappend substatusL [lindex $res 1]
      lappend selfstatusL [lindex $res 2]
      incr buds [lindex $res 2 0]
   }
```

In the fairly common case that there is only one child, there is no pruning to do and the statuses can all be reused, so we return early.

```
   if {[llength $childL] <= 1} then {
      lset res 0 [list node $attr $childL]
      return $res
   }
```

Otherwise check if there is pruning to do. When there is, the next step is to pick a subbranch to prune. The stochastic side is handled by building a list of $\sum_{l=1}^{k} x_l^q$ values, rescaling a random number to its range, and then doing a `-bisect` search to find the right range. Since this produces a linear index for the subbranch, it is also necessary to keep track of the branching system positions of each candidate subbranch, which is what the `posL` list is for.

```
   if {$limitsidx>=0 && $buds > [lindex $limitsL $limitsidx] &&\
                                                      rand()>=$p} then {
      set distL [list -Inf]
      set posL {}
      set sum 0.0
      set i 0
      foreach L $substatusL {
         set j 0
         foreach status $L {
            set sum [expr {$sum + pow(
              double([lindex $status 0])/[lindex $status 1], $q
            )}]
            lappend distL $sum
            lappend posL [list $i $j]
            incr j
         }
         incr i
      }
      set k\
            [lsearch -sorted -real -bisect $distL [expr {$sum * rand()}]]
      set i [lindex $posL $k 0]
      set j [lindex $posL $k 1]
```

i is the index of the child that will be subjected to pruning. j is the index at its first branching point of the subbranch to remove. This branching point may sit several levels down in the recursion, so `pruneIL` is constructed to hold the list of indices that will address the right list of children.

```
      if {[lindex $selfstatusL $i 0] > 1} then {
         set pruneIL [list $i 2]
         set subchildL [lindex $childL $pruneIL]
         while {[llength $subchildL] == 1} {
            lappend pruneIL 0 2
            set subchildL [lindex $subchildL 0 2]
         }
         set subchildL [lreplace $subchildL $j $j]
```

Before updating the actual tree (well, `childL`), we also update `buds` and `selfstatusL` accordingly. The latter either means decrement the bud count, or in the more common case that the node ceases to be a branching point, it means copy the entire status of the sole remaining subbranch.

```
         incr buds [expr {-[lindex $substatusL $i $j 0]}]
         if {[llength $subchildL] > 1} then {
            lset selfstatusL $i 0 [expr {
              [lindex $selfstatusL $i 0] - [lindex $substatusL $i $j 0]
            }]
         } else {
            lset selfstatusL $i [lindex $substatusL $i [expr {1-$j}]]
         }
         lset childL $pruneIL $subchildL
      } else {
```

Technically it could also happen that primary branch i had no branching points, which is the case precisely if it only has one bud (as checked above). This creates a similar distinction between the still-a-branching-point and the 2-to-1 children cases.

```
         set childL [lreplace $childL $i $i]
```

```
            if {[llength $childL] > 1} then {
               set selfstatusL [lreplace $selfstatusL $i $i]
            } else {
               set selfstatusL [lindex $substatusL [expr {1-$i}]]
            }
            incr buds -1
         }
      }
```

Now we are ready to return the result. The main thing that happens here is that the `selfstatusL` get demoted to {*subbranch-statuses*}, since this node was a branching point.

```
      return [list [list node $attr $childL] $selfstatusL\
                                 [list $buds [lindex $limitsL $limitsidx]]]
   }
```

`selfthin::up1n` (proc) The top-level `up1n` procedure has as one task to strip off the status parts returned by the bare operation, but it also performs an extra self-thinning step (the “root rule”). As usual, this consists of comparing a branch that joins the actual {*tree*} with a virtual twin against an equally virtual limit. The tricky part here is to determine the virtual limit.

The relevant data that will be available for free are the limits in the statuses returned by the recursive operation: one for the first branching point (“self”), and one each for every second branching point (“subs”). The idea is to extrapolate the change in limit from the subs c_i to the self c another step down, since that would be the limit at the virtual zeroth branching point. These limits are expected to follow a geometric sequence, so it seems reasonable that the scaling factor for “one step down” is something like

$$\frac{c}{\sqrt[k]{\prod_{i=1}^{k} c_i}},$$

i.e., the self limit divided by the geometric mean of the sub limits. The sought virtual limit is thus another c times this fraction.

Note at this point that it is possible for this virtual limit to actually be stricter than the one in force at the root, if the first and second branchings are close; this is like having the virtual branching above the actual root! If this happens we don’t do the virtual twin comparison, because we sort-of have actual data for that position and can see that there isn’t any twin branch there.

```
   proc selfthin::up1n {tree limitsL limitsidx p q} {
      foreach {tree substatusL selfstatus} [
         namespace inscope up1n $tree $limitsL $limitsidx $p $q
      ] break
```

Next check if there is additional pruning to do: obviously if the tree has no branching then there isn’t. We also do the random checks for ‘randomly skip pruning’ and ‘select actual or virtual branch for pruning’ before checking whether pruning would be triggered, since these are easy to combine and less work than doing the comparison.

```
      if {![llength $substatusL] || rand() < 0.5*(1-$p)} then\
                                                        {return $tree}
      set prod 1.0
      foreach status $substatusL {
         set prod [expr {$prod * [lindex $status 1]}]
      }
      set vlimit [expr\
         {[lindex $selfstatus 1]**2 / pow($prod,1.0/[llength $substatusL])}]
      if {$vlimit > [lindex $limitsL $limitsidx] &&\
                              2*[lindex $selfstatus 0] > $vlimit} then {
```

There will be pruning (in the actual branch), so we need to pick one of its sub-branches, with the right probability.

```
         set distL [list -Inf]
         set sum 0.0
         foreach status $substatusL {
            set sum [expr {$sum + pow(
              double([lindex $status 0])/[lindex $status 1], $q
            )}]
            lappend distL $sum
         }
         set k\
              [lsearch -sorted -real -bisect $distL [expr {$sum * rand()}]]
         set pruneIL [list 2]
         set subchildL [lindex $tree $pruneIL]
         while {[llength $subchildL] == 1} {
            lappend pruneIL 0 2
            set subchildL [lindex $subchildL 0 2]
         }
         lset tree $pruneIL [lreplace $subchildL $k $k]
      }
      return $tree
   }
```

4.4 Marking stress

A slight variation on the above thinning operation is the `markstress` operation which simply equips each node with an attribute specifying whether that node exceeds its limit or not. The call syntax is

`namespace inscope selfthin::markstress` {*tree*} {*limits-list*}
{*limits-index*} {*attrname*}

and the return value is a list

{*marked tree*} {*bud-count*}

As a transformation on the {*tree*}, the result is the {*marked tree*}. The {*bud-count*} is the number of buds in the tree (*not*, as above, lists of counts of buds in subtrees). The {*attrname*} is the name of the attribute to insert or replace; the value is `1` for stressed nodes and `0` for unstressed onces. The {*limits-list*} and {*limits-index*} are as above.

```
namespace eval selfthin::markstress {}
```

`selfthin::markstress::bud` (proc) Buds are assumed unstressed.

```
proc selfthin::markstress::bud {attr children limitsL limitsIdx name} {
   list {bud {} {}} 1
}
```

`selfthin::markstress::node` (proc) For a `node`, the first order of business is to recursively apply the operation to all children.

```
proc selfthin::markstress::node {attr children limitsL limitsidx name} {
   set childL {}
   set sum 0
   set nextidx [expr {$limitsidx-1}]
   foreach child $children {
      set res [{*}$child $limitsL $nextidx $name]
      lappend childL [lindex $res 0]
      incr sum [lindex $res 1]
   }
```

After that, one only needs to set the attribute and return the result.

```
   return [list [
      list node [dict replace $attr $name [expr {
         $limitsidx>=0 && $sum>[lindex $limitsL $limitsidx]
      }]] $childL
   ] $sum]
}
```

5 Data display and export

5.1 TDL

For a quick display format, TDL should do nicely.

`selfthin::to_TDL` (proc) This procedure has the call syntax

`selfthin::to_TDL` {*tree*} {*indent*}$^?$ {*prefix*}$^?$

and returns a TDL formatting of the {*tree*}. The {*indent*} is the indentation step (defaults to two spaces). The {*prefix*} is what should be put at the beginning of each line (defaults to the empty string, but used for surrounding indentation when recursing). The return value does not have a trailing newline.

```
proc selfthin::to_TDL {tree {indent "  "} {prefix ""}} {
   set res $prefix
   append res [lrange $tree 0 0]
   dict for {k v} [lindex $tree 1] {
      append res "  " [list $k $v]
   }
   if {[llength [lindex $tree 2]]} then {
      append res " \{\n"
      set last "$prefix\}"
      append prefix $indent
      foreach child [lindex $tree 2] {
         append res [to_TDL $child $indent $prefix] \n
      }
      append res $last
   }
   return $res
}
```

5.2 Matlab parent-list

A Matlab parent-list is a way of encoding a tree as a vector. Each element in the vector is the (1-based) index of the parent of the corresponding node. The tree root has a 0 in its element.

`selfthin::parentlist` (proc)

The call syntax is

`parentlist` {*list-var*} {*parent-index*} {*tree*} {*height*}

where {*list-var*} is the name in the calling context of a list to which parent-list elements for the {*tree*} will be appended. {*parent-index*} is the (1-based) index of the parent of the {*tree*}. {*height*} is how high up in the tree the recursion should continue. There is no particular return value.

```
proc selfthin::parentlist {listvar parent tree height} {
   upvar 1 $listvar res
   lappend res $parent
   if {[incr height -1] < 0} then {return}
   set me [llength $res]
   foreach child [lindex $tree 2] {
      parentlist res $me $child $height
   }
}
```

5.3 MetaPost code

The idea of the MetaPost code export alternative is to outright generate MetaPost code that draws the tree. The difficult part of that is to have the code generate sensible coordinates for the nodes, but the following approach seems to do the trick.

The idea is to have MetaPost compute all the coordinates, based on five pieces of information:

- A *last point*, which is where the current subtree is to be rooted.
- A *maximal* (leftmost) and *minimal* (rightmost) respectively *direction*, which define a sector with the *last point* as vertex. The idea is for the tree to be contained within that sector.
- The number of metamers to the next branching, and the number of metamers from that branching to the buds.

The distance from the last point to the next branching point is proportional to the number of metamers between them, and the direction is halfway between the leftmost and rightmost directions. Thereafter, the trick is to compute new maximal and minimal directions for the two children. This could just be taken as a bisection of the previous directions, but that tends to look a bit sparse and artificial. A more natural-looking result is obtained if one instead allows new sectors to have a *greater* angular extent than the parent, provided it is still small enough that the actual region the branch can reach does not increase; this latter calculation is what requires knowing the distance out to the buds.

The MP representation of a branch to draw is as a binary tree with explicit lengths of edges (where by length is meant the number of metamers between two nodes that actually are branching points). Concretely, the MP call syntax for drawing a branch is

⟨*branch*⟩ (*mcolour*, *fcolour*, *lastpoint*, *extra*, *maxdir*, *mindir*)

where the ⟨*branch*⟩ is what constitutes the representation of the branch, and the rest are extra parameters for that particular operation that need to be provided separately, typically passed on by each recursive call for drawing a subbranch. The *lastpoint*, *maxdir*, and *mindir* were described above, and are recalculated at each recursion step. The *extra* is an "extra number of metamers" that should be included in the length of the length of the root edge of the branch; its effect is to make the sector start further back, for example to account for the radius of the tree stem. The *fcolour* is the "foliage colour" to use for foliage (see below). The *mcolour*, finally, is the "marker colour" to use at marked nodes; such marks are used to signal that a node experiences stress due to crowding.

`selfthin::MP::macrodefs` (var.)

The definitions of the custom MP macros are stored in the Tcl variable `selfthin::MP::macrodefs` so that they can be output in a generated file. They can however also be docstripped out separately.

```
namespace eval selfthin::MP {variable macrodefs {
⟨/pkg⟩
⟨*pkg, MP⟩
```

For drawing buds (single bud branches), there is the **draw_bud** macro which does not take any extra parameters. It takes the edge length *steps* as its only parameter.

```
def draw_bud(expr steps)(
   expr markcolour, foliagecolour, last_point, extra, maxdir, mindir
 ) =
    draw last_point --
      last_point + steps*metamer * dir 0.5[maxdir,mindir];
enddef;
```

For drawing branching nodes, there is the **draw_node** macro. The representation of a branch of this kind is

> **draw_node(** *length*, *distance*, *marked?***)** **(**⟨*left*⟩**)** **(**⟨*right*⟩**)**

where ⟨*left*⟩ and ⟨*right*⟩ are again two branches. *length* is the number of metamers up to the first branching point, *distance* is the number of metamers that follow on the longest path after the branching point, and *marked?* is a boolean for whether this node should be marked.

```
  def draw_node(expr steps, tailsum, markme)(text left,right)(
     expr markcolour, foliagecolour, last_point, extra, maxdir, mindir
   )  =
      for mydir := 0.5[maxdir,mindir] :
      for me := last_point + steps*metamer * dir mydir :
         draw last_point -- me;
         for next_arc :=
            0.5(maxdir-mindir) +
            angle (tailsum, (extra+steps)*sind 0.5(maxdir-mindir))
```

This calculation of angular extent `next_arc` for subbranches is technically a bit wrong, but seems to produce servicable results, so why worry? (The error is that the angle corresponds to computing an arctan, where it strictly speaking should have been an arcsin for the new smaller sectors to reach all the way to the boundary of the old sector. The arctan does however have the nice feature that it avoids erroring out when it would be impossible to reach that far.)

Since the *extra* is to account for the stem, it is taken to be 0 in recursive calls.

```
         :
            left(markcolour, foliagecolour, me, 0, mydir+next_arc, mydir)
            right(markcolour, foliagecolour, me, 0, mydir, mydir-next_arc)
         endfor
         if markme:
            fill fullcircle scaled 0.4metamer shifted me
              withcolor markcolour;
         fi
      endfor endfor
  enddef;
```

An alternative to generating a huge number of nodes in the branch periphery is however to encode sufficiently small subbranches as "foliage". The representation of a branch of this kind is

> **draw_foliage(** *length*, *distance*, *marked?***)**

where *length* is the number of metamers up to the first branching point, *distance* is the number of metamers that follow on the longest path after the branching point, and *marked?* is a boolean for whether this node should be marked.

Implementation-wise, this kind of branch has a lot in common with a metamer node, but instead of having individual children, the whole circle sector region allocated for such children is simply filled in with a solid colour. Unlike the case for **draw_node**, the base metamer is drawn after children.

```
def draw_foliage(expr steps, tail, markme)(
   expr markcolour, foliagecolour, last_point, extra, maxdir, mindir
 )  =
    for mydir := 0.5[maxdir,mindir] :
    for me := last_point + steps*metamer * dir mydir :
       for next_arc :=
          0.5(maxdir-mindir) +
          angle (tail, (extra+steps)*sind 0.5(maxdir-mindir))
       :
          filldraw me -- (me + tail*metamer*dir(mydir+next_arc))
            {dir (mydir+next_arc-90)} .. {dir (mydir-next_arc-90)}
            (me + tail*metamer*dir(mydir-next_arc)) -- cycle
            withcolor foliagecolour;
       endfor
       draw last_point -- me;
       if markme:
          fill fullcircle scaled 0.4metamer shifted me
            withcolor markcolour;
       fi
    endfor endfor
enddef;
```
⟨/pkg, MP⟩
⟨∗pkg⟩
```
}}
```

The data-is-code operation for generating such a MetaPost representation of a tree is not entirely straightforward, since there is a need for communicating data both up and down the tree. The call syntax is

⟨*tree*⟩ {*length*} {*markattr*} {*thickness*} {*indent*}

where {*length*} is the accumulated length of metamers since the last branching point, {*markattr*} is the name of the node attribute to look at for deciding whether a node should be marked, {*thickness*} is the maximal foliage thickness (measured from bud to foliage branching point), and {*indent*} is the current indentation level for generated code. The return value from that has the syntax

{*code*} {*distance*}

where {*code*} is the generated MP code (typically not with newline first or last, but with indentation on each line, including the first) and {*distance*} is {*length*} more than the length of the longest path from the root of the {*tree*}. The latter has the effect that it is precisely the *distance* needed for a parent **draw_node**.

`selfthin::MP::bud` (proc) Buds are still pretty easy to implement this operation for.

```
proc selfthin::MP::bud\
                      {attrD children length markattr thickness indent} {
```

```
      incr length
      return [list "${indent}draw_bud(${length})" $length]
   }
```

selfthin::MP::node (proc) General nodes have two distinct cases depending on the number of children. If there is a single child, then just delegate generating any code to that child (incrementing the {*length*} by one):

```
   proc selfthin::MP::node\
                          {attrD children length markattr thickness indent} {
      incr length
      if {[llength $children] == 1} then {
         return\
              [{*}[lindex $children 0] $length $markattr $thickness $indent]
      } elseif {[llength $children] == 2} then {
```

The main case is however that there are two children.

```
         set left\
                 [{*}[lindex $children 0] 0 $markattr $thickness "$indent "]
         set right\
                 [{*}[lindex $children 1] 0 $markattr $thickness "$indent "]
         set mark [expr {
            [dict exists $attrD $markattr] && [dict get $attrD $markattr]
            ? "true" : "false"
         }]
         set max [expr {max([lindex $left 1],[lindex $right 1])}]
         if {$max <= $thickness} then {
            set code [format {%sdraw_foliage(%d,%d,%s)} $indent $length\
                                                                  $max $mark]
         } else {
            set code\
                 [format {%sdraw_node(%d,%d,%s)} $indent $length $max $mark]
            append code "(\n" [lindex $left 0] "\n$indent)(\n"\
                                         [lindex $right 0] "\n$indent)"
         }
         return [list $code [expr {$max+$length}]]
      } else {
```

Better guard against arities greater than 2.

```
         return -code error "This operation only supports nodes with 1 or\
                                                                  2 children"
      }
   }
```

selfthin::to_MP (proc) The operation is obscure enough that it can do with a wrapper. This has the call syntax

   `selfthin::to_MP` {*tree*} {*markattr*} {*thickness*} {*indent*}

and returns the generated code.

```
   proc selfthin::to_MP {tree markattr thickness indent} {
      lindex [
         namespace inscope MP $tree 0 $markattr $thickness $indent
      ] 0
   }
```

`selfthin::lateral_branch_fig` (proc)

This is a higher level wrapper that returns code for a full **beginfig–endfig** block. The call syntax is

`selfthin::lateral_branch_fig` {*fignum*} {*tree*} $\bigl($\{*option*\} \{*value*\}$\bigr)^*$

where {*tree*} is the tree to make a figure out of, and {*fignum*} is the sequence number given to that figure. The recognised {*option*}s are:

-markattr The name of the mark attribute. Defaults to `stress`.

-maxdir The maximal direction (number of degrees counterclockwise from positive x-axis) of the full branch sector, i.e., its left edge. Defaults to `120`.

-mindir The minimal direction of the full branch sector, i.e., its right edge. Defaults to `60`.

-metamer The length of a metamer, in `cm` (unit not included in value). Defaults to `0.2`.

-stem The radius of the stem, in `cm` (unit not included in value). Defaults to `1.0`.

-markcol The **color** to use for marked nodes. Defaults to `red`.

-stemcol The **color** to use for shading the stem (or rather, the *sector* of the stem included in the figure). If not given, then the stem is not **fill**ed.

-foliagecol The **color** to use for foliage. Defaults to `green`.

-foliage The {*thickness*} of the foliage, in growth cycles. Defaults to `-1` (definitely no foliage).

-branchpen The **pen** to use for drawing the branch. Defaults to `pencircle\ scaled 0.1metamer`.

-stempen The **pen** to use for drawing the stem. Defaults to `pencircle scaled\ 0.2metamer`.

```
proc selfthin::lateral_branch_fig {num tree args} {
   array set O {
      -maxdir  120
      -mindir   60
      -metamer   0.2
      -stem      1.0
      -markcol    red
      -foliagecol green
      -foliage    -1
      -branchpen  {pencircle scaled 0.1metamer}
```

```
         -stempen    {pencircle scaled 0.2metamer}
         -markattr   stress
      }
      array set O $args
      set res [format {beginfig(%d);} $num]
      append res "\n  metamer := $O(-metamer)cm;"
      append res "\n  stem := $O(-stem)cm;"
      append res "\n  pickup $O(-branchpen);"
      append res \n [
         to_MP $tree $O(-markattr) $O(-foliage) "  "
      ] [format {(%s, %s, (0,0), %.2f, %s, %s)} $O(-markcol)\
                  $O(-foliagecol) [expr {double($O(-stem))/$O(-metamer)}]\
                                                 $O(-maxdir) $O(-mindir)]
      set fullarc [expr {$O(-maxdir)-$O(-mindir)}]
      set halfarc [expr {0.5*$fullarc}]
      if {[info exists O(-stemcol)]} then {
         append res "\n  fill ((0,0) -- (stem,0){up} .."
         append res "\n    stem*dir$halfarc{dir$halfarc zscaled (0,1)} .."
         append res "\n    {dir$fullarc zscaled (0,1)}stem*dir$fullarc"
         append res "\n    -- cycle) shifted (-stem*dir$halfarc) rotated\
                                                             $O(-mindir)"
         append res "\n    withcolor $O(-stemcol);"
      }
      append res "\n  draw ((stem,0){up} .."
      append res "\n    stem*dir$halfarc{dir$halfarc zscaled (0,1)} .."
      append res "\n    {dir$fullarc zscaled (0,1)}stem*dir$fullarc"
      append res "\n    ) shifted (-stem*dir$halfarc) rotated $O(-mindir)"
      append res "\n    withpen $O(-stempen);"
      append res \n "endfig;"
      return $res
   }
```

A variant application of the MetaPost tree representation would be to draw "level curves" in the tree. Here, it is instead more practical to let the entire representation of a branch expand to a ⟨*future path*⟩ for one level curve. Since the default branch representation instead aims to draw the branch, it would be necessary to (temporarily) redefine the **draw_**(**bud**|**node**|**foliage**) macros, but that's something one can do using **let** if the alternative definitions are available. These can preferably be named **levelcurve_**...

The MP call syntax for getting a level curve for a branch is thus

⟨*branch*⟩ (*level*, *lastpoint*, *extra*, *maxdir*, *mindir*)

where the *level* is the level (in growth cycles, but not necessarily an integer) at which to sample the ⟨*branch*⟩, and the rest are as for the drawing operation. The above should expand to zero or more repetitions of

⟨*point*⟩ ..

where each ⟨*point*⟩ is some point where the level curve intersects the branch.

Buds are quite easy, but should demonstrate the general pattern.

⟨/pkg⟩
⟨*MP⟩

```
def levelcurve_bud(expr steps)(
   expr level, last_point, extra, maxdir, mindir
 ) =
    if level <= steps:
       (last_point + level*metamer * dir 0.5[maxdir,mindir])
       ..
    fi
enddef;
```

Branching nodes expand on the above to perform recursion over left and right subbranches, but only if the *level* is above what this node covers.

```
def levelcurve_node(expr steps, tailsum, markme)(text left,right)(
   expr level, last_point, extra, maxdir, mindir
 )  =
   for mydir := 0.5[maxdir,mindir] :
   if level <= steps:
      (last_point + level*metamer * dir mydir)
      ..
   else:
      for me := last_point + steps*metamer * dir mydir :
      for next_arc :=
         0.5(maxdir-mindir) +
         angle (tailsum, (extra+steps)*sind 0.5(maxdir-mindir))
      :
         left(level-steps, me, 0, mydir+next_arc, mydir)
         right(level-steps, me, 0, mydir, mydir-next_arc)
      endfor endfor
   fi endfor
enddef;
```

Foliage nodes are similar, but instead of recursing there is a three-point representation for the foliage: left edge, middle, and right edge.

```
def levelcurve_foliage(expr steps, tail, markme)(
   expr level, last_point, extra, maxdir, mindir
 )  =
    for mydir := 0.5[maxdir,mindir] :
    if level <= steps:
      (last_point + level*metamer * dir mydir)
      ..
    else: if level <= steps+tail:
       for me := last_point + steps*metamer * dir mydir :
       for next_arc :=
          0.5(maxdir-mindir) +
          angle (tail, (extra+steps)*sind 0.5(maxdir-mindir))
       :
          (me + (level-steps)*metamer * dir(mydir+next_arc))
          ..
          (me + (level-steps)*metamer * dir(mydir))
          ..
          (me + (level-steps)*metamer * dir(mydir-next_arc))
          ..
       endfor endfor
    fi fi
    endfor
enddef;
```
⟨/MP⟩
⟨∗pkg⟩

6 Putting it all together

`selfthin::repeatcycles` (proc) This procedure repeats a number of grow–prune cycles on a tree, and returns the resulting tree. The call syntax is

`selfthin::repeatcycles` {*tree*} {*height*} {*cycles*} {*mu*} {*alpha*} {*dimension*}

where {*tree*} is the input tree, {*height*} is the height of that tree (affects the size bound), {*cycles*} is the number of cycles to run, {*mu*} is the branching factor when growing the tree, {*alpha*} is the constant in the bound formula, and {*dimension*} is the exponent in that formula.

The bound formula is that in repetition m there must be at most $\alpha(m+h-k)^d$ buds above a node at level k from the root.

```
proc selfthin::repeatcycles {tree height cycles mu alpha d} {
   set limitL {}
   for {set n 1} {$n <= $height+$cycles+1} {incr n} {
      lappend limitL [expr {$alpha*pow($n,$d)}]
   }
   for {set n 0} {$n < $cycles} {incr n} {
      set tree [lindex [
         namespace inscope up1d [
            namespace inscope grow $tree $mu
         ] $limitL $height $n
      ] 0]
      incr height
   }
   return $tree
}
```

`selfthin::growto` (proc) The `growto` procedure takes a bud and lets it grow some number of cycles, while performing a `up1r` self-thinning step after each cycle. The call syntax is

`selfthin::growto` {*height*} {*mu*} {*alpha*} {*dimension*} {*ignore-probability*}

where {*height*} is the target height, {*mu*} is the branching factor when growing the tree, {*alpha*} is the constant in the bound formula, {*dimension*} is the exponent in that formula, and {*ignore-probability*} is the probability for not enforcing the size bound at a particular node and cycle. The return value is the resulting tree.

```
proc selfthin::growto {height mu alpha d p} {
   set limitL {}
   for {set n 1} {$n <= 2*$height+1} {incr n} {
      lappend limitL [expr {$alpha*pow($n,$d)}]
   }
   set tree [list bud {} {}]
   for {set n 0} {$n < $height} {incr n} {
⟨*!Poisson⟩
      set tree [up1r [namespace inscope grow $tree $mu] $limitL $n $p]
⟨/!Poisson⟩
⟨*Poisson⟩
      set tree [up1r_Poisson [namespace inscope grow $tree $mu] $limitL\
                                                              $n $p $d]
⟨/Poisson⟩
   }
   return $tree
}
```

`selfthin::rgrowto` (proc) The `rgrowto` procedure takes a bud and lets it `rgrow` some number of cycles, while performing a `up1r_Poisson` self-thinning step after each cycle. The call syntax is

`selfthin::rgrowto` {*height*} {*distribution*} {*alpha*} {*dimension*} {*ignore-probability*}

where {*height*} is the target height, {*mu*} is the branching factor when growing the tree, {*alpha*} is the constant in the bound formula, {*dimension*} is the exponent in that formula, and {*ignore-probability*} is the probability for not enforcing the size bound at a particular node and cycle. The return value is the resulting tree.

```
proc selfthin::rgrowto {height distribution alpha d p} {
   set limitL {}
   for {set n 1} {$n <= 2*$height+1} {incr n} {
      lappend limitL [expr {$alpha*pow($n,$d)}]
   }
   set tree [list bud {} {}]
   for {set n 0} {$n < $height} {incr n} {
      set tree [up1r_Poisson [
         namespace inscope rgrow $tree $distribution
      ] $limitL $n $p $d]
   }
   return $tree
}
```

`selfthin::ngrowto` (proc) The `ngrowto` procedure takes a bud and lets it `rgrow` some number of cycles, while performing a `up1n` self-thinning step after each cycle. The call syntax is

`selfthin::rgrowto` {*height*} {*distribution*} {*alpha*} {*dimension*} {*ignore-probability*} {*weight-exponent*} where {*height*} is the target height, {*distribution*} is the distribution for the primary branching factor when growing the tree, {*alpha*} is the constant in the bound formula, {*dimension*} is the exponent in that formula, {*ignore-probability*} is the probability for not enforcing the size bound at a particular node and cycle, and {*weight-exponent*} is how much the pruning probability is weighted towards branches exceeding carrying capacity. The return value is the resulting tree.

```
proc selfthin::ngrowto {height distribution alpha d p q} {
   set limitL {}
   for {set n 1} {$n <= 2*$height+1} {incr n} {
      lappend limitL [expr {$alpha*pow($n,$d)}]
   }
   set tree [list bud {} {}]
   for {set n 1} {$n <= $height} {incr n} {
      set tree [up1n [
         namespace inscope rgrow $tree $distribution
      ] $limitL $n $p $q]
   }
   return $tree
}
```

`selfthin::growth_progression` (proc)

This procedure grows a tree/branch a number of steps while generating code for MetaPost figures of each step. The call syntax is

`selfthin::growth_progression` {*channel*} {*cycles*} $\bigl($ {*option*} {*value*} $\bigr)^*$

where the {*channel*} is the channel to which the MP code should be written; this will be a complete valid source file, including all necessary macro definitions and final **end**. The {*cycles*} is the number of growth cycles to generate. There are two figures for each growth cycle: before `grow` and before thinning. There is also an extra figure for the final state. The final state (as a list-tree) is also the return value from this procedure.

There are a large number of options, in particular:

-starttree The data-tree to use as starting point. This defaults to a single bud.

-alpha The constant factor in the geometric saturation bound. Defaults to `3.0`.

-dimension The dimensionality of the branch sector. Defaults to `1`, and since the figures generated put the whole branch in a single plane that is probably the only value that makes visual sense. There should however not be a logical problem with using other values.

-thinmethod One of `up1d`, `up1r_Poisson`, and `up1r`. Defaults to `up1r`.

-seed Seed value for the thinning, must be an integer, defaults to `22035`. In `up1d` thinning this is precisely the {*seed*} value, and it is incremented between growth cycles. In `up1r` thinning it is used once to initialise the Tcl built-in `rand()` function. (Always actively setting a seed is good when one wants to start finetuning the generated figures.)

-ignorep The {*ignore-probability*} for `up1r` thinning. Defaults to `0.1`.

-markcol0 The mark colour to use in pre-`grow` figures. Defaults to `blue`.

-markcol1 The mark colour to use in pre-thinning figures. Defaults to `red`.

All options and values are furthermore passed on to `lateral_branch_fig`, although some of those may be overridden in the call.

One thing that is noticably *not* an option is the μ value. This is fixed at 2.

```
 proc selfthin::growth_progression {F cycles args} {
    array set O {
       -starttree  {bud {} {}}
       -alpha      3.0
       -dimension  1
       -thinmethod up1r
       -seed       22035
       -ignorep    0.1
       -markcol0   blue
       -markcol1   red
    }
    array set O $args
    if {$O(-thinmethod) eq "up1r"} then {
       tcl::mathfunc::srand $O(-seed)
    }
```

The first step is to write a preamble to the channel.

```
    puts $F "% MetaPost code generated by selfthin::growth_progression."
    puts $F [set [namespace current]::MP::macrodefs]
```

Then the limit-list is calculated. `height` will serve as starting limit-index.

```
    set tree $O(-starttree)
    set height [llength [namespace inscope histogram $tree]]
    set limitL {}
    for {set n 1} {$n <= 2*($height+$cycles+1)} {incr n} {
       lappend limitL [expr {$O(-alpha)*pow($n,$O(-dimension))}]
    }
```

Then there's the main loop over the growth cycles.

```
    set figno 0
    for {incr cycles -1} {$cycles >= 0} {incr cycles -1} {
       puts $F [
          lateral_branch_fig [incr figno] [lindex [
             namespace inscope markstress $tree $limitL $height stress
          ] 0] {*}$args -markcol $O(-markcol0)
       ]
       set tree [namespace inscope grow $tree 2]
       incr height
       puts $F [
          lateral_branch_fig [incr figno] [lindex [
             namespace inscope markstress $tree $limitL $height stress
          ] 0] {*}$args -markcol $O(-markcol1)
```

```
      ]
      switch -- $O(-thinmethod) "up1r" {
         set tree [up1r $tree $limitL $height $O(-ignorep)]
      } "up1r_Poisson" {
         set tree [up1r_Poisson $tree $limitL $height $O(-ignorep)\
                                                         $O(-dimension)]
      } "up1d" {
         lassign [
            namespace inscope up1d $tree $limitL $height $O(-seed)
         ] tree counts
```

The `up1d` method should get an extra oversaturation check at the root, just like the `up1r` procedure does.

```
         if {[llength $counts]>1 && [::tcl::mathop::+ {*}$counts] >\
                                           [lindex $limitL $height]} then {
            set pruneIL [list 2]
            set subchildL [lindex $tree $pruneIL]
            while {[llength $subchildL] == 1} {
               lappend pruneIL 0 2
               set subchildL [lindex $subchildL 0 2]
            }
            set j [expr {$O(-seed) % [llength $subchildL]}]
            lset tree $pruneIL [lreplace $subchildL $j $j]
         }
         incr O(-seed)
      }
   }
```

Finally there's the last figure and postamble.

```
   puts $F [
      lateral_branch_fig [incr figno] [lindex [
         namespace inscope markstress $tree $limitL $height stress
      ] 0] {*}$args -markcol $O(-markcol0)
   ]
   puts $F "end;"
   return $tree
}
```

⟨/pkg⟩

7 Logging in database

The following is about running a simulation and storing the branches produced in a database. So first we need TDBC.

⟨∗database⟩
```
package require tdbc
package require tdbc::sqlite3
```

`selfthin::new_database` (proc)

The `new_database` procedure creates a database connection (which could involve creating the database file in the first place) and makes sure it contains a table named `br4nches`. The call syntax is

`selfthin::new_database` {*db-cmd*} {*db-file*} {*width*}

where {*db-cmd*} is name to assign to the database connection, {*db-file*} is the name of the database file, and {*width*} is the number of `g`⟨*k*⟩ columns (starting with `g0`) created in the `br4nches` table. Apart from these, there is also a column `a6e` which holds the age of the full plant.

```
proc selfthin::new_database {dbcmd file width} {
   tdbc::sqlite3::connection create $dbcmd $file
   set sql {CREATE TABLE IF NOT EXISTS br4nches ( a6e INTEGER}
   incr width 0
   for {set k 0} {$k < $width} {incr k} {
      append sql ", g$k INTEGER"
   }
   append sql {)}
   $dbcmd allrows $sql
}
```

`selfthin::prepare_database` (proc)

This procedure creates two commands that perform prepared statements on a database. The call syntax is

`prepare_database` {*database*} {*insert-command*} {*count-command*}

where {*database*} is the name of the database connection object whereas {*insert-command*} and {*count-command*} are the names of two commands that will be created by this procedure. The first has the call syntax

{*insert-command*} {*plant-age*} {*histogram*}

and inserts one row into the `br4nches` table of the database. The {*plant-age*} goes into the `a6e` column, whereas {*histogram*} is a list whose elements go into the `g`⟨*k*⟩ columns (each *k* being equal to the index into this list). The second command has the call syntax

{*count-command*} {*plant-age*}

and returns the number of rows in the table which have the specified `a6e`. Both commands are implemented as aliases to an `apply` with one presupplied argument that is the relevant prepared-statement object.

```
proc selfthin::prepare_database {db insertcmd countcmd} {
```

Beginning with the {*countcmd*}, which is slightly simpler (fixed and explicit SQL code), one may observe that this can benefit greatly from an index on the `a6e` column, so let's add one if it isn't there already.

```
   $db allrows {CREATE INDEX IF NOT EXISTS br4nchesA6e ON br4nches(a6e)}
   interp alias {} $countcmd {} apply {{stmt age} {
      lindex [$stmt allrows -as lists [dict create age $age]] 0 0
   } ::} [$db prepare {SELECT count(*) FROM br4nches WHERE a6e = :age}]
```

As for the {*insertcmd*}, it is not obvious from the given procedure arguments is how many parameters the prepared statement will have, but this information is available from introspection into the database.

```
   set width [dict size [$db columns br4nches g%]]
   set colL [list a6e]
   for {set k 0} {$k < $width} {incr k} {lappend colL g$k}
   set stmt [$db prepare "INSERT INTO br4nches ([join $colL ,]) VALUES\
                                                    (:[join $colL ,:])"]
   interp alias {} $insertcmd {} apply {{stmt age histogram} {
      set D [dict create]
      foreach h $histogram {dict set D g[dict size $D] $h}
      dict set D a6e $age
      $stmt allrows $D
   } ::} $stmt
}
⟨/database⟩
```

`selfthin::insert_samples` (proc) This procedure generates at least a given number of samples of branches at a given height in a plant of a given age, and inserts these samples into a database. The call syntax is

> `insert_samples` {*insert-cmd*} {*samples-goal*} {*sample-height*} {*plant-age*} {*generate-cmd*}

and there is no particular return value. The {*insert-cmd*} is the command used to insert samples. The {*samples-goal*} is the minimal number of samples to insert; since samples are generated in batches, and no part of a batch is thrown away, the number of samples inserted may exceed the goal. The {*sample-height*} is the height (distance from root) at which samples are taken. The {*generate-cmd*} is the command called to grow one tree; classically it would be

> `growto` {*plant-age*} {*mu*} {*alpha*} {*dimension*} {*ignore-probability*}

Obviously {*plant-age*} has to be larger than {*sample-height*}, if one is to get any nontrivial results.

```
proc selfthin::insert_samples {inscmd goal height age gencmd} {
   while {$goal > 0} {
      set tree [{*}$gencmd]
      foreach branch [
         if {$height > 0} then {
            namespace inscope harvest $tree $height
         } else {
            list $tree
         }
      ] {
         $inscmd $age [namespace inscope histogram $branch]
         incr goal -1
      }
   }
}
```

`selfthin::params` (var.) This variable holds the list of values for the {*mu*}, {*alpha*}, {*dimension*}, and {*ignore-probability*} parameters.

```
namespace eval selfthin {
   variable params {2 3.0 2 0.1}
}
```

`selfthin::sample_iterator` (proc) This procedure is a basic event loop iterator for the process of gathering samples. It has the call syntax

> `selfthin::sample_iterator` {*inscmd*} {*countcmd*} {*goal*} {*length*} {*ages*} {*subgoal*}$^?$ {*ageindex*}$^?$

where {*goal*} is the number of samples to collect for each plant age being sampled, {*length*} is the length of the branches to collect, {*ages*} is the list of ages to gather data for, {*subgoal*} is (if given and {*ageindex*} is nonzero) the goal to strive for in this iteration, and {*ageindex*} (which defaults to `0`) is the index in the {*ages*} list of the class to work at in this iteration.

```
proc selfthin::sample_iterator\
          {inscmd countcmd goal length ages {subgoal -1} {ageindex 0}} {
   variable params
   if {$ageindex} then {
      set age [lindex $ages $ageindex]
      set n [expr {$subgoal - [$countcmd $age]}]
      insert_samples $inscmd $n [expr {$age-$length}] $age\
                                       [list growto $age {*}$params]
      incr ageindex
   } else {
      set age [lindex $ages 0]
      insert_samples $inscmd 1 [expr {$age-$length}] $age\
                                       [list growto $age {*}$params]
      set subgoal [$countcmd $age]
      set ageindex 1
⟨stdout⟩      puts "Subgoal $subgoal at [clock format [clock seconds]]."
   }
   if {$ageindex >= [llength $ages]} then {set ageindex 0}
   after idle [list after 1 [
      list [namespace which sample_iterator] $inscmd $countcmd $goal\
                                    $length $ages $subgoal $ageindex
   ]]
}
```

```
⟨*database⟩
⟨*GUI⟩
package require Tk
pack [label .params_l -text "Parameters (mu alpha d p):"] -side top\
                                                              -fill x
pack [entry .params_e -textvariable ::selfthin::params] -side top -fill\
                                                                    x
pack [button .run_b -text "Run" -state disabled -command {
   ::selfthin::sample_iterator ::insert_row ::count_by_age 1000000 10\
                                                        {100 50 20 15 10}
}] -side right
pack [button .stop_b -text "Stop" -command {
   ::apply {jobs {
      foreach id $jobs {after cancel $id}
   } ::} [after info]
}] -side right
pack [button .set_b -text "Apply" -command {
   selfthin::new_database ::thedb\
                            branches-[join $::selfthin::params -].db 10
   selfthin::prepare_database ::thedb ::insert_row ::count_by_age
   .run_b configure -state normal
}] -side left
⟨/GUI⟩
⟨*GUI20130122⟩
package require Tk
pack [label .params_l -text "Parameters (mu alpha d p):"] -side top\
                                                                  -fill x
pack [entry .params_e -textvariable ::selfthin::params] -side top -fill\
                                                                       x
pack [button .run_b -text "Run" -state disabled -command {
   ::selfthin::sample_iterator ::insert_row ::count_by_age 100 100 {100}
}] -side right
pack [button .stop_b -text "Stop" -command {
   ::apply {jobs {
      foreach id $jobs {after cancel $id}
   } ::} [after info]
}] -side right
pack [button .set_b -text "Apply" -command {
   selfthin::new_database ::thedb\
                          wholetree-[join $::selfthin::params -].db 100
   selfthin::prepare_database ::thedb ::insert_row ::count_by_age
   .run_b configure -state normal
}] -side left
⟨/GUI20130122⟩
```

The following is a minimal alternative control mechanism: Create a child that sits waiting forever, which when killed will initiate a controlled shutdown of the present process.

```
⟨*noTk20130311⟩
set kill_to_exit_child [open "|[info nameofexecutable]" r+]
fconfigure $kill_to_exit_child -buffering none -blocking no
puts $kill_to_exit_child "vwait forever\n"
fileevent $kill_to_exit_child readable "
   catch {[list ::close $kill_to_exit_child]}
   set ::forever 0
"
set L [lassign $argv goal age]
if {[llength $L] != 4} then {
   puts stderr "Parameters: goal age mu alpha d p"
   exit
}
set selfthin::params $L
selfthin::new_database ::thedb\
                     wholetree=$age-[join $::selfthin::params -].db $age
selfthin::prepare_database ::thedb ::insert_row ::count_by_age
set forever 1
while {$forever} {
   selfthin::insert_samples ::insert_row 1 0 $age\
                                 [list growto $age {*}$selfthin::params]
   if {[::count_by_age $age] >= $goal} then {break}
   update
}
::thedb close
catch {
   exec kill [pid $kill_to_exit_child]
}
exit
⟨/noTk20130311⟩
⟨/database⟩
 —

⟨*simplerun20170510⟩
set age 100
set silviculture\
              [list ::selfthin::rgrowto $age {-Inf 0 0.2 0.8} 3.0 2 0.1]
set F [open wholetree-rgrow-[
   join [lrange $silviculture end-2 end] -
].csv w]
interp alias {} insert_row {} apply {{F age histogram} {
   puts $F "$age,[join $histogram ,]"
}} $F
selfthin::insert_samples ::insert_row 100 0 100 $silviculture
close $F
⟨/simplerun20170510⟩
⟨*simplerun20170513⟩
set age 100
foreach distribution {
   {-Inf 0 0} {-Inf 0 0.2 0.8}
} dname {
   fixed random
} {
   foreach weighting {0 1 2} {
      set silviculture [list ::selfthin::ngrowto $age $distribution 3.0\
                                                      2 0.1 $weighting]
      set F [open wholetree-$dname-[
         join [lrange $silviculture end-3 end] -
      ].csv w]
      fconfigure $F -buffering line
      interp alias {} insert_row {} apply {{F age histogram} {
         puts $F "$age,[join $histogram ,]"
      }} $F
      selfthin::insert_samples ::insert_row 200 0 100 $silviculture
      close $F
   }
}
```
⟨/simplerun20170513⟩

8 Figure scripts

8.1 Branching loss under growth

⟨∗branchgrowfig⟩
```
set alpha 2.0
set d 1
set limitL {}
for {set n 1} {$n <= 20} {incr n} {
   lappend limitL [expr {$alpha*pow($n,$d)}]
}
set mu 2
set Tree(0) {bud {id /} {}}
for {set n 0} {$n < 15} {incr n} {
   set Tree([expr {$n+1}]) [
      selfthin::up1n [
         namespace inscope selfthin::igrow $Tree($n) $mu
      ] $limitL [expr {$n+1}] 0.1 0
   ]
}
foreach age {5 8 11} {
   set ETree($age) [
      selfthin::markwhere $Tree($age) id [list ::dict exists [
         dict filter [selfthin::aritybyid $Tree([expr {$age+3}])] value\
                                                                       2
      ]] extant 1
   ]
}
set res [selfthin::lateral_branch_fig 50 $ETree(5) -maxdir 30 -mindir\
                         -30 -foliage 1 -markattr extant -markcol blue]
append res \n [selfthin::lateral_branch_fig 51 $ETree(8) -maxdir 30\
                  -mindir -30 -foliage 1 -markattr extant -markcol blue]
append res \n [selfthin::lateral_branch_fig 52 $ETree(11) -maxdir 30\
                  -mindir -30 -foliage 1 -markattr extant -markcol blue]
append res \n [selfthin::lateral_branch_fig 53 $Tree(14) -maxdir 30\
                  -mindir -30 -foliage 1 -markattr extant -markcol blue]
set PTree(5) $Tree(5)
for {set n 5} {$n < 15} {incr n} {
   set PTree([expr {$n+1}])\
                        [namespace inscope selfthin::grow $PTree($n) 2]
}
append res \n [selfthin::lateral_branch_fig 54 $PTree(11) -maxdir 30\
                                                   -mindir -30 -foliage 1]
```

⟨/branchgrowfig⟩

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.