A step-by-step guide to writing a simple package that uses S4 methods: a "hello world" example

Robin K. S. Hankin

January 8, 2007

1 Introduction

This vignette proves that it is possible for a 'normal' person[1] to write a package using S4 methods. It gives a step-by-step guide to creating a package that contains two S4 classes (brobs and glubs) and a bunch of basic utilities for manipulating them.

This document focuses on the S4 aspects of the package. For an overview of the mathematical properties of Brobdingnagian numbers, their potential and their limitations, see the `.Rd` files and the entry in Rnews (forthcoming).

If you like this vignette and package, and find it useful, let me know. If there is anything wrong with it, let me know.

I would not recommend that anyone uses S4 unless there is a good reason for it (many of my packages use S3 methods which I found to be perfectly adequate for my needs). Reasons for using S4 might include a package having a large number of object classes that have a complicated hierarchical structure, or a complicated set of methods that interact with the object classes in a complicated manner.

In the package, `glub` objects are treated in `glub.R` which appropriately generalizes all the `brob` functionality.

This document could not have been prepared (and should not be read) without consulting the following resources:

- ^ John M. Chambers (1998), *Programming with Data*. New York: Springer, ISBN 0-387-98503-4 (The Green book).
- ^ W. N. Venables and B. D. Ripley (2000), *S Programming*. Springer, ISBN 0-387-98966-8.
- ^ John Chambers (2006). `How S4 methods work` (available on CRAN).

1.1 Overview

The idea of `Brobdingnag` package is stunningly simple: the IEEE representation for floating point numbers cannot represent numbers larger than about 1.8×10^{308}. The package represents a number by the natural logarithm of its magnitude, and also stores a Boolean value indicating its sign. Objects so stored have class `brob`; complex numbers may be similarly represented and have class `glub`.

With this scheme, multiplication is easy but addition is hard. The basic identity is:

$$\log(e^x + e^y) = \begin{cases} x + \log(1 + e^{y-x}) & \text{if } x > y \\ y + \log(1 + e^{x-y}) & \text{otherwise} \end{cases}$$

In practice this gets more complicated as one has to keep track of the sign; and special dispensation is needed for zero ($= e^{-\infty}$).

[1] That is, someone without super powers (such as might manifest themselves after being bitten by a radioactive member of R-core, for example)

One can thus deal with numbers up to about $e^{1.9\times10^{308}} \simeq 10^{7.8\times10^{307}}$, although at this outer limit accuracy is pretty poor.

2 Class definition

OK, let's get going. The first thing we need to do is to define the class. This uses the `setClass()` function:

```
> setClass("swift", representation = "VIRTUAL")
> setClass("brob", representation = representation(x = "numeric",
+     positive = "logical"), prototype = list(x = numeric(), positive = logical()),
+     contains = "swift")
```

It is simpler to ignore the first call to `setClass()` here; for reasons that will become apparent when discussing the `c()` function, one needs a virtual class that contains class brob and glub. To understand virtual classes, see section 15.

The second call to `setClass()` is more germane. Let's take this apart, argument by argument. The first argument, `representation`, specifies "the slots that the new class should have and/or other classes that this class extends. Usually a call to the 'representation' function". The helppage for `representation` gives a few further details. Thus this argument specifies two 'slots': one for the value and one for the sign. These are specified to be numeric and logical (not Boolean) respectively.

The second argument, `prototype`, specifies default data for the slots. This kicks in when defining a zero-length brob; an example would be extracting `x[FALSE]` where `x` is a brob.

The third argument, `contains`, tells R that class `swift` (which was specified to be virtual), has `brob` as a subclass. We will need this later when we start to deal with `glub`s, which are also a subclass of `swift`.

OK, let's use it:

```
> new("brob", x = 1:10, positive = rep(TRUE, 10))

An object of class "brob"
Slot "x":
 [1]  1  2  3  4  5  6  7  8  9 10

Slot "positive":
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Notes:

- Function `new()` is the *only* way to create objects of class brob. So, any object of class brob *must* have been created with function `new()`. This is part of what the "formal" tag for S4 means[2].
- Function `new()` requires its arguments to be named, and no partial argument matching is performed.
- Function `new()` is not intended for the user. It's too picky and difficult. To create new brobs, we need some more friendly functions—`as.brob()` and `brob()`—discussed below.
- There is, as yet, no print method, so the form of the object printed to the screen is less than ideal.

[2]Compare S3, in which I can say `a <- 1:10; class(a) <- "lilliput"`

2.1 Validity methods

Now, an optional step is to define a function that tests whether the arguments passed to `new()` are acceptable. As it stands, the following code:

```
> new("brob", x = 1:10, positive = c(TRUE, FALSE, FALSE))

An object of class "brob"
Slot "x":
 [1]  1  2  3  4  5  6  7  8  9 10

Slot "positive":
[1]  TRUE FALSE FALSE
```

will not return an error, but is not acceptable because the arguments are different lengths (and will not recycle neatly).

So, we define a validity method:

```
> .Brob.valid <- function(object) {
+     len <- length(object@positive)
+     if (len != length(object@x)) {
+         warning("length mismatch")
+         return(FALSE)
+     }
+     else {
+         return(TRUE)
+     }
+ }
```

In this package, I define a whole bunch of functions whose name starts with `.Brob.`; these are internal and not intended for the user. They are also not documented.

OK, so now we have a function, `.Brob.valid()`, that checks whether its argument has slots of the same length. We need to tell R that this function should be invoked every time a `brob` is created. Function `setValidity()` does this:

```
> setValidity("brob", .Brob.valid)

Slots:

Name:         x positive
Class:  numeric  logical

Extends: "swift"
```

Thus, from now on [ie after the above call to `setValidity()`], when calling `new("brob", ...)` the two arguments `x` and `positive` must be the same length: recycling is not carried out.

Functions like `.Brob.valid()` that are user-unfriendly all have names beginning with `.Brob`. These functions are there to halp the organization of the package and are not intended to be used by the end-user.

Clever, user-friendly operations such as recycling are carried out in the more user-friendly functions such as `as.brob()`.

OK, just to check, let's call `new()` with arguments of differing lengths:

```
> try(new("brob", x = 1:10, positive = TRUE))

[1] "Error in validObject(.Object) : invalid class \"brob\" object: FALSE\n"
attr(,"class")
[1] "try-error"
```

So `new()` throws a wobbly (the warning message issued by `.Brob.valid()` is printed in a session but does not appear here) because the `positive` argument had length 1 and the `x` was length 10; and the validity method `.Brob.valid()` requires both arguments to be the same length.

So that works, but isn't exactly user-friendly. This is remedied in the next section.

3 Basic user-friendly functions to create brobs

The basic, semi user-friendly function for creating brobs is `brob()`:

```
> "brob" <- function(x = double(), positive) {
+     if (missing(positive)) {
+         positive <- rep(TRUE, length(x))
+     }
+     if (length(positive) == 1) {
+         positive <- rep(positive, length(x))
+     }
+     new("brob", x = as.numeric(x), positive = positive)
+ }
```

Thus `brob(x)` will return e^x. Function `brob()` does helpful things like assuming the user desires positive numbers; it also carries out recycling:

```
> brob(1:10, FALSE)

An object of class "brob"
Slot "x":
 [1]  1  2  3  4  5  6  7  8  9 10

Slot "positive":
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Note that `brob()` isn't exactly terribly user-friendly: it's confusing. `brob(5)` returns a number formally equal to e^5, not 5. This is documented in the help page, where the user is encouraged to use function `as.brob()` instead.

4 Testing for brobs: an `is.brob()` function

The next thing we need is an `is.brob()` function:

```
> is.brob <- function(x) {
+     is(x, "brob")
+ }
> is.glub <- function(x) {
+     is(x, "glub")
+ }
```

We also define an `is.glub()` function. Note that the user-friendly function `is.brob()` calls the user-unfriendly `is()` function of the `methods` package.

So now we can check for objects being brobs and glubs.

5 Coercion: an `as.brob()` function

Next, some ways to coerce stuff to brob form:

```
> "as.brob" <- function(x) {
+     if (is.brob(x)) {
+         return(x)
+     }
+     else if (is.complex(x)) {
+         warning("imaginary parts discarded")
+         return(Recall(Re(x)))
+     }
+     else if (is.glub(x)) {
+         warning("imaginary parts discarded")
+         return(Re(x))
+     }
+     else {
+         return(brob(log(abs(x)), x >= 0))
+     }
+ }
```

So now we can coerce stuff to brobs. The *only* way to create a brob is to use `new()`, and the *only* function that calls this is `brob()`. And `as.brob()` calls this.

Note the user-friendliness of `as.brob()`. It takes numerics, brobs, and glubs (which give a warning).

Better check it:

```
> as.brob(1:10)

An object of class "brob"
Slot "x":
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851

Slot "positive":
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

6 Coersion: an `as.numeric()` function

Now we need some methods to coerce brobs to numeric. This is a two-stage process.

```
> setAs("brob", "numeric", function(from) {
+     out <- exp(from@x)
+     out[!from@positive] <- -out[!from@positive]
+     return(out)
+ })
```

This call to `setAs()` makes `as(x,"numeric")` carry out the function passed as the third argument when given a brob.

But the user isn't supposed to type `as(x,"numeric")`: the user is supposed to type `as.numeric(x)`. To accomplish this, we have to tell R that function `as.numeric()` should execute `as(x,"numeric")` when given a `brob`. This is done by calling function `setMethod()`:

```
> setMethod("as.numeric", signature(x = "brob"), function(x) {
+     as(x, "numeric")
+ })
```

We similarly need to make `as.complex()` work for brobs:

```
> setAs("brob", "complex", function(from) {
+     return(as.numeric(from) + (0+0i))
+ })
> setMethod("as.complex", signature(x = "brob"), function(x) {
+     as(x, "complex")
+ })
```

We'll need similar methods for glubs too.
Better check:

```
> x <- as.brob(1:4)
> x

An object of class "brob"
Slot "x":
[1] 0.0000000 0.6931472 1.0986123 1.3862944

Slot "positive":
[1] TRUE TRUE TRUE TRUE

> as.numeric(x)

[1] 1 2 3 4
```

So that works.

7 Print methods

OK, now some print methods. First, a helper function:

```
> .Brob.print <- function(x, digits = 5) {
+     noquote(paste(c("-", "+")[1 + x@positive], "exp(", signif(x@x,
+         digits), ")", sep = ""))
+ }
```

Then an S3 method:

```
> print.brob <- function(x, ...) {
+     jj <- .Brob.print(x, ...)
+     print(jj)
+     return(invisible(jj))
+ }
```

And finally a call to `setMethod()`:

```
> setMethod("show", "brob", function(object) {
+     print.brob(object)
+ })
```

This two-stage methodology is recommended in Programming with Data. The `.Brob.print()` function does the hard work. Example of it in use:

```
> as.brob(1:4)

[1] +exp(0)       +exp(0.69315) +exp(1.0986)  +exp(1.3863)
```

See how the brob object is printed out nicely, and with no special effort required of the user.

8 Get and Set methods

To be anal retentive about things, one should define C++ style accessor functions as follows:

```
> setGeneric("getX", function(x) {
+     standardGeneric("getX")
+ })
> setGeneric("getP", function(x) {
+     standardGeneric("getP")
+ })
> setMethod("getX", "brob", function(x) {
+     x@x
+ })
> setMethod("getP", "brob", function(x) {
+     x@positive
+ })
```

but in practice I just use `@` to access the slots. These are just here for good form's sake.

9 Length

Now a length:

```
> setMethod("length", "brob", function(x) {
+     length(x@x)
+ })
```

10 Extracting elements of a vector

Next thing is to define some methods for extraction. This is done with `setMethod()` for extraction, and `setReplaceMethod()` for replacement.

```
> setMethod("[", "brob", function(x, i, j, drop) {
+     if (!missing(j)) {
+         warning("second argument to extractor function ignored")
+     }
+     brob(x@x[i], x@positive[i])
+ })
```

See how the third argument to `setMethod()` is a function whose arguments are the same as those to `"["()` . Argument `j` *must* be there otherwise one gets a signature error. I've put in a warning if a second argument that might be interpreted as `j` is given.

OK, now a method for replacement. This is a call to `setReplaceMethod()`:

```
> setReplaceMethod("[", signature(x = "brob"), function(x, i, j,
+     value) {
+     if (!missing(j)) {
+         warning("second argument to extractor function ignored")
+     }
+     jj.x <- x@x
+     jj.pos <- x@positive
+     if (is.brob(value)) {
+         jj.x[i] <- value@x
+         jj.pos[i] <- value@positive
```

```
+         return(brob(x = jj.x, positive = jj.pos))
+     }
+     else {
+         x[i] <- as.brob(value)
+         return(x)
+     }
+ })
```

See how the replacement function tests for the replacement value being a brob and acts accordingly.

11 Concatenation function `cbrob()`

Right, next thing. It is not possible to make `c()` behave as expected for brobs[3] (that is, if any of its arguments are brobs, to coerce all its arguments to brobs and then concatenate).

However, it is possible to define a function `cbrob()` that does the job. This has to be done in several stages.

First we define another user-unfriendly helper function `.Brob.cPair()` which takes two arguments, coerces them to brobs, and concatenates them:

```
> .Brob.cPair <- function(x, y) {
+     x <- as.brob(x)
+     y <- as.brob(y)
+     brob(c(x@x, y@x), c(x@positive, y@positive))
+ }
```

This is just `c()` for the two slots separately. The idea is that function `.Brob.cPair()` takes two arguments; both are coerced to brobs and it returns the concatenated vector of brobs.

Now, we need to set up a (user-unfriendly) generic function `.cPair()`:

```
> setGeneric(".cPair", function(x, y) {
+     standardGeneric(".cPair")
+ })
```

Function `.cPair()` is not substantive (sic): it exists purely in order to be a generic function that dispatches to `.Brob.cPair()`.

Now we use `setMethod()` to organize the dispatch:

```
> setMethod(".cPair", c("brob", "brob"), function(x, y) {
+     .Brob.cPair(x, y)
+ })
> setMethod(".cPair", c("brob", "ANY"), function(x, y) {
+     .Brob.cPair(x, as.brob(y))
+ })
> setMethod(".cPair", c("ANY", "brob"), function(x, y) {
+     .Brob.cPair(as.brob(x), y)
+ })
> setMethod(".cPair", c("ANY", "ANY"), function(x, y) {
+     c(x, y)
+ })
```

[3]The ideas in this section are entirely due to John Chambers, who kindly replied to a question of mine on the R-devel email list

The four calls are necessary for the four different signatures that might be encountered. Note the `ANY` class in the second, third, and fourth call. Thus if someone wants to write a new class of object (a lugg, say), and wants to concatenate luggs with a brob, this will work provided that they use `setAs()` to make `as.brob()` coerce correctly for lugg objects. The method used here allows this to be done without any changes to the Brobdingnag package.

OK, the final stage: definition of `cbrob()`, a user-friendly wrapper for the above stuff:

```
> "cbrob" <- function(x, ...) {
+     if (nargs() < 3)
+         .cPair(x, ...)
+     else .cPair(x, Recall(...))
+ }
```

Note the recursive definition. If `cbrob()` is called with *any* set of arguments that include a brob anywhere, this will result in the whole lot being coerced to brobs [by `.Brob.cPair()`]. Which is what we want (although glubs will require more work).

Just test this:

```
> a <- 1:3
> b <- as.brob(1e+100)
> cbrob(a, a, b, a)

 [1] +exp(0)       +exp(0.69315) +exp(1.0986)  +exp(0)       +exp(0.69315)
 [6] +exp(1.0986)  +exp(230.26)  +exp(0)       +exp(0.69315) +exp(1.0986)
```

So it worked: everything was coerced to a brob because of the single object of class brob in the call.

12 Maths

There are two gotchas here: firstly, `log()` needs to be made a generic with `setGeneric()`, and secondly, `sqrt()` needs a specific brob method.

```
> setGeneric("log", group = "Math")

> setMethod("sqrt", "brob", function(x) {
+     brob(ifelse(x@positive, x@x/2, NaN), TRUE)
+ })
```

Just check that:

```
> sqrt(brob(4))

[1] +exp(2)
```

With these out of the way we can use `setMethod()` to define the appropriate functions in the math group generic:

```
> setMethod("Math", "brob", function(x) {
+     switch(.Generic, abs = brob(x@x), log = {
+         out <- x@x
+         out[!x@positive] <- NaN
+         out
+     }, exp = brob(x), cosh = {
+         (brob(x) + brob(-x))/2
+     }, sinh = {
```

```
+         (brob(x) - brob(-x))/2
+     }, acos = , acosh = , asin = , asinh = , atan = , atanh = ,
+         cos = , sin = , tan = , tanh = , trunc = callGeneric(as.numeric(x)),
+         lgamma = , cumsum = , gamma = , ceiling = , floor = as.brob(callGeneric(as.numeric(x)))
+         stop(paste(.Generic, "not allowed on Brobdingnagian numbers")))
+ })
```

See how the third argument to `setMethod()` is a function. This function has access to `.Generic`, in addition to `x` and uses it to decide which operation to perform.

See how functions `acos()` to `trunc()` just drop through to `callGeneric(as.numeric(x))`. See also the method for `log()`, which uses facts about brobs not known to S4.

Just a quick check:

```
> sin(brob(4))

[1] -0.928768
```

So that works.

13 Operations

Now we need to make sure that `brob(1) + brob(3)` works: the operations `+, -, *, /` must work as expected. This is hard.

First step: define some user-unfriendly functions that carry out the operations. For example, function `.Brob.negative()` simply returns the negative of a brob. These functions are not for the user.

```
> .Brob.negative <- function(e1) {
+     brob(e1@x, !e1@positive)
+ }
> .Brob.ds <- function(e1, e2) {
+     xor(e1@positive, e2@positive)
+ }
> .Brob.add <- function(e1, e2) {
+     e1 <- as.brob(e1)
+     e2 <- as.brob(e2)
+     jj <- rbind(e1@x, e2@x)
+     x1 <- jj[1, ]
+     x2 <- jj[2, ]
+     out.x <- double(length(x1))
+     jj <- rbind(e1@positive, e2@positive)
+     p1 <- jj[1, ]
+     p2 <- jj[2, ]
+     out.pos <- p1
+     ds <- .Brob.ds(e1, e2)
+     ss <- !ds
+     out.x[ss] <- pmax(x1[ss], x2[ss]) + log1p(+exp(-abs(x1[ss] -
+         x2[ss])))
+     out.x[ds] <- pmax(x1[ds], x2[ds]) + log1p(-exp(-abs(x1[ds] -
+         x2[ds])))
+     out.x[(x1 == -Inf) & (x2 == -Inf)] <- -Inf
+     out.pos <- p1
+     out.pos[ds] <- xor((x1[ds] > x2[ds]), (!p1[ds]))
+     return(brob(out.x, out.pos))
```

```
+ }
> .Brob.mult <- function(e1, e2) {
+     e1 <- as.brob(e1)
+     e2 <- as.brob(e2)
+     return(brob(e1@x + e2@x, !.Brob.ds(e1, e2)))
+ }
> .Brob.power <- function(e1, e2) {
+     stopifnot(is.brob(e1) | is.brob(e2))
+     if (is.brob(e2)) {
+         return(brob(log(e1) * brob(e2@x), TRUE))
+     }
+     else {
+         s <- as.integer(2 * e1@positive - 1)
+         return(brob(e1@x * as.brob(e2), (s^as.numeric(e2)) >
+             0))
+     }
+ }
> .Brob.inverse <- function(b) {
+     brob(-b@x, b@positive)
+ }
```

Note the complexity of `.Brob.add()`. This is hard because logs are good at multiplying but bad at adding [`ss` and `ds` mean "same sign" and "different sign" respectively].

The first step is to make sure that the unary operators `+` and `-` work. We do this by a call to `setMethod()`:

```
> setMethod("Arith", signature(e1 = "brob", e2 = "missing"), function(e1,
+     e2) {
+     switch(.Generic, "+" = e1, "-" = .Brob.negative(e1), stop(paste("Unary operator",
+         .Generic, "not allowed on Brobdingnagian numbers")))
+ })
```

Note the second argument is `signature(e1 = "brob", e2="missing")`: this effectively restricts the scope to unary operators. The `switch()` statement only allows the + and the -.

Check:

```
> -brob(5)

[1] -exp(5)
```

So that works.

Next step, another user-unfriendly helper function that does the dirty work:

```
> .Brob.arith <- function(e1, e2) {
+     switch(.Generic, "+" = .Brob.add(e1, e2), "-" = .Brob.add(e1,
+         .Brob.negative(as.brob(e2))), "*" = .Brob.mult(e1, e2),
+         "/" = .Brob.mult(e1, .Brob.inverse(as.brob(e2))), "^" = .Brob.power(e1,
+             e2), stop(paste("binary operator \"", .Generic, "\" not defined for Brobdingnagian
+ }
```

And now we can call `setMethod()`:

```
> setMethod("Arith", signature(e1 = "brob", e2 = "ANY"), .Brob.arith)
> setMethod("Arith", signature(e1 = "ANY", e2 = "brob"), .Brob.arith)
> setMethod("Arith", signature(e1 = "brob", e2 = "brob"), .Brob.arith)
```

Better check it:

```
> 1e+100 + as.brob(10)^100

[1] +exp(230.95)
```

14 Comparison

Well this is pretty much the same as the others. First, some user-unfriendly helper functions:

```
> .Brob.equal <- function(e1, e2) {
+     (e1@x == e2@x) & (e1@positive == e2@positive)
+ }
> .Brob.greater <- function(e1, e2) {
+     jj.x <- rbind(e1@x, e2@x)
+     jj.p <- rbind(e1@positive, e2@positive)
+     ds <- .Brob.ds(e1, e2)
+     ss <- !ds
+     greater <- logical(length(ss))
+     greater[ds] <- jj.p[1, ds]
+     greater[ss] <- jj.p[1, ss] & (jj.x[1, ss] > jj.x[2, ss])
+     return(greater)
+ }
```

These are the fundamental ones. We can now define another all-encompassing user-unfriendly function:

```
> .Brob.compare <- function(e1, e2) {
+     e1 <- as.brob(e1)
+     e2 <- as.brob(e2)
+     switch(.Generic, "==" = .Brob.equal(e1, e2), "!=" = !.Brob.equal(e1,
+         e2), ">" = .Brob.greater(e1, e2), "<" = !.Brob.greater(e1,
+         e2), ">=" = .Brob.greater(e1, e2) | .Brob.equal(e1, e2),
+         "<=" = !.Brob.greater(e1, e2) | .Brob.equal(e1, e2),
+         stop(paste(.Generic, "not supported for Brobdingnagian numbers")))
+ }
```

See how this function coerces both arguments to brobs. Now the call to `setMethod()`:

```
> setMethod("Compare", signature(e1 = "brob", e2 = "ANY"), .Brob.compare)
> setMethod("Compare", signature(e1 = "ANY", e2 = "brob"), .Brob.compare)
> setMethod("Compare", signature(e1 = "brob", e2 = "brob"), .Brob.compare)
```

Better check:

```
> as.brob(10) < as.brob(11)

[1] TRUE

> as.brob(10) <= as.brob(10)

[1] TRUE
```

So that works.

15 Logic

(Note that the material in this section works in R-2.4.1, but not R-2.4.0).

First a helper function:

```
> .Brob.logic <- function(e1, e2) {
+     stop("No logic currently implemented for Brobdingnagian numbers")
+ }
```

Now the calls to `setMethod()`:

```
> setMethod("Logic", signature(e1 = "swift", e2 = "ANY"), .Brob.logic)
> setMethod("Logic", signature(e1 = "ANY", e2 = "swift"), .Brob.logic)
> setMethod("Logic", signature(e1 = "swift", e2 = "swift"), .Brob.logic)
```

Note that the signatures specify `swift` objects, so that glubs will be handled correctly too in one fell swoop. Note the third call to `setMethod()`: without this, a call to `Logic()` with signature `c("swift","swift")` would be ambiguous as it might be interpreted as `c("swift","ANY")` or `c("ANY","swift")`.

Here, class `swift` extends `brob` and `glub`; so both brobs and glubs *are* `swift` objects. But no object is a "pure" `swift`; it's either a brob or a glub. This is useful here because I might dream up some new class of objects that are "like" brobs in some way (for example, a class of objects whose elements are quaternions with Brobdingnagian components) and it would be nice to specify behaviour that is generic to brobs and glubs and the new class of objects too.

16 Miscellaneous generics

We now have to tell R that certain functions are to be considered generic. The functions are `max()`, `min()`, `range()`, `prod()`, and `sum()`. The help page for (eg) `max()` specifies that the arguments must be numeric, and brobs aren't numeric.

This is done by function `setGeneric()`; see the manpage. The manpage discusses the `useAsDefault` argument, which will stop a non-generic function from being used (which in this case is a Bad Thing).

OK, here goes:

```
> setGeneric("max", function(x, ..., na.rm = FALSE) {
+     standardGeneric("max")
+ }, useAsDefault = function(x, ..., na.rm = FALSE) {
+     base::max(x, ..., na.rm = na.rm)
+ }, group = "Summary")
> setGeneric("min", function(x, ..., na.rm = FALSE) {
+     standardGeneric("min")
+ }, useAsDefault = function(x, ..., na.rm = FALSE) {
+     base::min(x, ..., na.rm = na.rm)
+ }, group = "Summary")
> setGeneric("range", function(x, ..., na.rm = FALSE) {
+     standardGeneric("range")
+ }, useAsDefault = function(x, ..., na.rm = FALSE) {
+     base::range(x, ..., na.rm = na.rm)
+ }, group = "Summary")
> setGeneric("prod", function(x, ..., na.rm = FALSE) {
+     standardGeneric("prod")
+ }, useAsDefault = function(x, ..., na.rm = FALSE) {
+     base::prod(x, ..., na.rm = na.rm)
```

```
+ }, group = "Summary")
> setGeneric("sum", function(x, ..., na.rm = FALSE) {
+     standardGeneric("sum")
+ }, useAsDefault = function(x, ..., na.rm = FALSE) {
+     base::sum(x, ..., na.rm = na.rm)
+ }, group = "Summary")
```

Now we need some more user-unfriendly helper functions:

```
> .Brob.max <- function(x, ..., na.rm = FALSE) {
+     p <- x@positive
+     val <- x@x
+     if (any(p)) {
+         return(brob(max(val[p])))
+     }
+     else {
+         return(brob(min(val), FALSE))
+     }
+ }
> .Brob.prod <- function(x) {
+     p <- x@positive
+     val <- x@x
+     return(brob(sum(val), (sum(p)%%2) == 0))
+ }
> .Brob.sum <- function(x) {
+     .Brob.sum.allpositive(x[x > 0]) - .Brob.sum.allpositive(-x[x <
+         0])
+ }
> .Brob.sum.allpositive <- function(x) {
+     if (length(x) < 1) {
+         return(as.brob(0))
+     }
+     val <- x@x
+     p <- x@positive
+     mv <- max(val)
+     return(brob(mv + log1p(sum(exp(val[-which.max(val)] - mv))),
+         TRUE))
+ }
```

Note the final function that sums its arguments, which are all assumed to be positive, in an intelligent, accurate, and efficient manner. No checking is done (this is not a user-friendly function!).

We can define `.Brob.sum()` in terms of this: return the difference between the sum of the positive arguments and and the sum of minus the negative arguments.

```
> setMethod("Summary", "brob", function(x, ..., na.rm = FALSE) {
+     switch(.Generic, max = .Brob.max(x, ..., na.rm = na.rm),
+         min = -.Brob.max(-x, ..., na.rm = na.rm), range = cbrob(min(x,
+             na.rm = na.rm), max(x, na.rm = na.rm)), prod = .Brob.prod(x),
+         sum = .Brob.sum(x), stop(paste(.Generic, "not allowed on Brobdingnagian numbers")))
+ })
```

Better check it:

```
> sum(as.brob(1:100)) - 5050
```

```
[1] -exp(-Inf)
```

Not bad.

17 Examples of the package in use

OK, let's try factorials. Stirlings approximation is $n! \sim \sqrt{2\pi n}\, e^{-n} n^n$:

```
> stirling <- function(x) {
+     sqrt(2 * pi * x) * exp(-x) * x^x
+ }
```

And this should work seamlessly with Brobs:

```
> stirling(100)

[1] 9.324848e+157

> stirling(as.brob(100))

[1] +exp(363.74)
```

And are they the same?

```
> as.numeric(stirling(100)/stirling(as.brob(100)))

[1] 1
```

... pretty near. But where IEEE flakes out, Brobs wade in:

```
> stirling(1000)

[1] NaN

> stirling(as.brob(1000))

[1] +exp(5912.1)
```

And this is accurate to about 12 sig figs, which is accurate enough for my purposes. The number of sig figs decreases with progressively larger numbers, essentially because increasing amounts of floating point accuracy is gobbled up by storing the exponent of a large number, and less is left for the mantissa.

Robin K. S. Hankin
National Oceanography Centre, Southampton
European Way
Southampton SO14 3ZH
United Kingdom
E-mail: r.hankin@noc.soton.ac.uk URL: http://www.noc.soton.ac.uk