# Overview of the *intervals* package

Richard Bourgon

08 September 2008

## Contents

**1 Introduction** **1**

**2 Interpretation of objects** **2**
2.1 As a subset of $\mathbb{Z}$ or $\mathbb{R}$ . . . . . 3
2.2 As a set of meaningful, possibly overlapping intervals . . . . . 4

**3 Floating point and intervals over $\mathbb{R}$** **6**

**4 Notes on implementation** **8**
4.1 Endpoint representation . . . . . 8
4.2 Efficiency . . . . . 9
4.3 Checking validity . . . . . 9

**5 Session information** **10**

## 1 Introduction

The *intervals* packages defines two S4 classes which represent collections of intervals over either the integers ($\mathbb{Z}$) or the real number line ($\mathbb{R}$). An instance of either class consists of a two-column matrix of endpoints, plus additional slots describing endpoint closure and whether the intervals are to be thought of as being over $\mathbb{Z}$ or $\mathbb{R}$.

```
> library( intervals )
> x <- Intervals( matrix( 1:6, ncol = 2 ) )
> x

Object of class Intervals
3 intervals over R:
[1, 4]
[2, 5]
[3, 6]

> x[2,2] <- NA
> x[3,1] <- 6
> x
```

```
Object of class Intervals
3 intervals over R:
[1, 4]
[2, NA]
[6, 6]
```

Objects of class `Intervals` represent collections of intervals with common endpoint closure, e.g., all left-closed, right-open. More control over endpoints is permitted with the `Intervals_full` class. (Both classes are derived from `Intervals_virtual`, which is not intended for use by package users.)

```
> y <- as( x, "Intervals_full" )
> closed(y)[2:3,1] <- FALSE
> y

Object of class Intervals_full
3 intervals over R:
[1, 4]
(2, NA]
(6, 6]
```

The `size` method gives measure — counting measure over $\mathbb{Z}$ or Lebesgue measure over $\mathbb{R}$ — for each interval represented in an object. The `empty` method identifies intervals that are in fact empty sets, which over $\mathbb{R}$ is not the same thing as having size 0. (Valid objects must have each right endpoint no less than the corresponding left endpoint. When one or both endpoints are open, however, valid intervals may be empty.)

```
> size(x)

[1]  3 NA  0

> empty(x)

[1] FALSE    NA FALSE

> empty(y)

[1] FALSE    NA  TRUE
```

# 2 Interpretation of objects

An `Intervals` or `Intervals_full` object can be thought of in two different modes, each of which is useful in certain contexts:

1. As a (non-unique) representation of a subset of $\mathbb{Z}$ or $\mathbb{R}$.
2. As a collection of (possibly overlapping) intervals, each of which has a meaningful identity.

## 2.1 As a subset of $\mathbb{Z}$ or $\mathbb{R}$

The *intervals* package provides a number of basic tools for working in the first mode, where an object represents a subset of $\mathbb{Z}$ or $\mathbb{R}$ but the rows of the endpoint matrix do not have any external identity. Basic tools include `reduce`, which returns a sorted minimal representation equivalent to the original (dropping any intervals with `NA` endpoints), as well as `interval_union`, `interval_complement`, and `interval_intersection`.

```
> reduce( y )

Object of class Intervals_full
1 interval over R:
[1, 4]

> interval_intersection( x, x + 2 )

Object of class Intervals
2 intervals over R:
[3, 4]
[6, 6]

> interval_complement( x )

Object of class Intervals
3 intervals over R:
(-Inf, 1)
(4, 6)
(6, Inf)
```

Note that combining `x` and its complement in order to compute a union requires mixing endpoint closure types; coercion to `Intervals_full` is automatic.

```
> interval_union( x, interval_complement( x ) )

Object of class Intervals_full
1 interval over R:
(-Inf, Inf)
```

The `distance_to_nearest` function treats its `to` argument in the first mode, as just a subset of $\mathbb{Z}$ or $\mathbb{R}$; it treats its `from` argument, however, in the second mode, returning one distance for every row of the `from` object. In the example below, we also look at performance for large data sets (less than one second on a 2 GHz Intel Core 2 Duo Macintosh, although the time shown below will likely differ). A histogram of `d` is given in Figure 1.

```
> B <- 100000
> left <- runif( B, 0, 1e8 )
> right <- left + rexp( B, rate = 1/10 )
> v <- Intervals( cbind( left, right ) )
> head( v )

Object of class Intervals
6 intervals over R:
[60223931.0974255, 60223939.4771043]
```

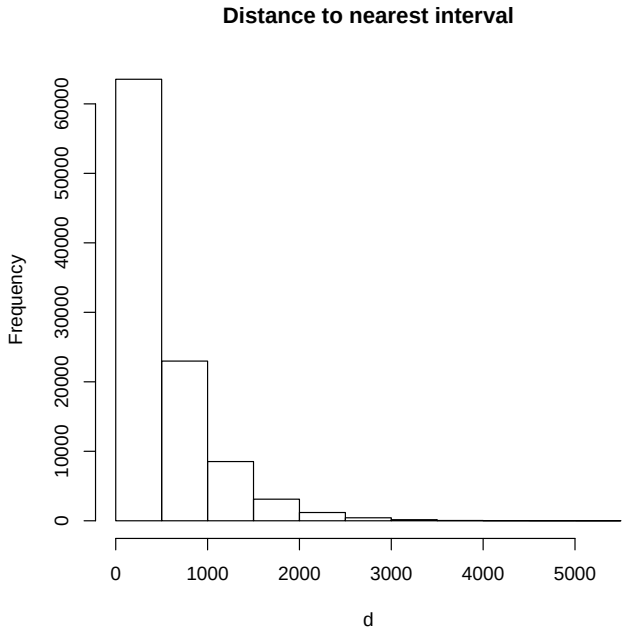


Figure 1: Histogram of distances from a random set of points to the nearest point in `v`. There is also a `distance_to_nearest` method for comparing two sets of intervals.

```
[92588142.7014247, 92588147.952952]
[82099781.1155394, 82099784.323378]
[47450154.623948, 47450162.5124498]
[66365371.5979308, 66365374.7654459]
[24596364.3072173, 24596377.2869091]

> mean( size( v ) )

[1] 9.986575

> dim( reduce( v ) )

[1] 98977     2

> system.time( d <- distance_to_nearest( sample( 1e8, B ), v ) )

   user  system elapsed
  0.798   0.710   1.512
```

## 2.2 As a set of meaningful, possibly overlapping intervals

In some applications, each row of an object's endpoint matrix has a meaningful identity, and particular points from $\mathbb{Z}$ or $\mathbb{R}$ may be found in more than one row. To support this mode, objects may be given row names, which are propagated

through calculations when appropriate. The `combine` methods simply stack objects (like `rbind`), preserving row names and retaining redundancy, if any.

The `interval_overlap` method works in this mode. In the next example we use it to identify rows of `v` which are at least partially redundant, i.e., which intersect at least one other row of `v`. All rows overlap themselves, so we look for rows that overlap at least two rows:

```
> rownames(v) <- sprintf( "%06i", 1:nrow(v) )
> io <- interval_overlap( v, v )
> head( io, n = 3 )

$`000001`
[1] 1

$`000002`
[1] 2

$`000003`
[1] 3

> n <- sapply( io, length )
> sum( n > 1 )

[1] 2032

> k <- which.max( n )
> io[ k ]

$`003605`
[1] 11236 63834  3605

> v[ k, ]

Object of class Intervals
1 interval over R:
003605 [41543963.3419737, 41543969.8531428]

> v[ io[[ k ]], ]

Object of class Intervals
3 intervals over R:
011236 [41543957.5212076, 41543963.9014843]
063834 [41543960.5712891, 41543970.6195700]
003605 [41543963.3419737, 41543969.8531428]
```

The `which_nearest` method (v. 0.9.10, *not yet implemented*) also respects row identity, returning a list with one vector of indices into the `to` object for each row of the `from` object.

Another function which operates in this mode is `clusters`, which takes a set of points or intervals and identifies maximal groups which cluster together — which are separated from one another by no more than a user-specified threshold. The following code is taken from the `clusters` documentation:

```
> B <- 100
> left <- runif( B, 0, 1e4 )
> right <- left + rexp( B, rate = 1/10 )
> y <- reduce( Intervals( cbind( left, right ) ) )
> w <- 100
> c2 <- clusters( y, w )
> c2[1:3]

[[1]]
Object of class Intervals
3 intervals over R:
[43.3801161125302, 43.9790035598683]
[45.7966956309974, 66.2250471204044]
[113.896930124611, 115.878578829579]

[[2]]
Object of class Intervals
3 intervals over R:
[263.932205270976, 273.423710493371]
[327.473098877817, 338.808666318235]
[379.641084000468, 414.217079205621]

[[3]]
Object of class Intervals
2 intervals over R:
[587.36790670082, 600.735664339316]
[665.032791439444, 694.232040901933]
```

# 3 Floating point and intervals over $\mathbb{R}$

When `type == "R"`, interval endpoints are not truly in $\mathbb{R}$, but rather, in the subset which can be represented by floating point arithmetic. (For the moment, this is also true when `type == "Z"`. See Section 4.1.) This limits the endpoint values which can be represented; more importantly, if computations are performed on interval endpoints, it means that floating point error can affect whether or not endpoints coincide, whether intervals which meet at or near endpoints overlap one another, etc.

In spite of this potentially serious limitation, it is still often convenient to work with intervals with non-integer endpoints, including data where adjacent intervals exactly meet at a non-integer endpoint. To address this, the *intervals* package takes the following approach:

- Floating point representations of interval endpoints are assumed to be *exactly equal* (in the sense of `==` in R or C++) if and only if the user intends the real values corresponding to these representations to be exactly equal.
- For cases where floating point error and approximate equality are a concern, tools are provided to permit distinguishing between ambiguous and unambiguous intersection, union, etc.

In the next example, `y1` does not literally overlap `y2[2,]`, although R's

all.equal function asserts that the gap between them is smaller than the default tolerance for equivalence up to floating point precision.

```
> delta <- .Machine[[ "double.eps" ]]^0.5
> y1 <- Intervals( c( .5, 1 - delta / 2 ) )
> y2 <- Intervals( c( .25, 1, .75, 2 ) )
> y1

Object of class Intervals
1 interval over R:
[0.5, 0.99999999254942]

> y2

Object of class Intervals
2 intervals over R:
[0.25, 0.75]
[1, 2]

> all.equal( y1[1,2], y2[2,1] )

[1] TRUE

> interval_intersection( y1, y2 )

Object of class Intervals
1 interval over R:
[0.5, 0.75]
```

The expand and contract methods, used with type = "relative", permit consideration of the maximal and minimal interval sets which are consistent with the nominal endpoints — from the point of view of endpoint relative difference. The contract method, for example, contracts each interval in a collection so that the relative difference between original and contracted endpoints is equal to tolerance delta. Thus, if a relative difference less than or equal to delta is our criterion for approximate floating point equality, the contracted object has endpoints which are approximately equal to those of the original — even though the contracted object is a proper subset of the original. The expand method is similar, but generates a proper superset.

```
> contract( y1, delta, "relative" )

Object of class Intervals
1 interval over R:
[0.500000007450581, 0.999999977648258]
```

We compute two separate intersections which bound the nominal intersection:

```
> inner <- interval_intersection(
+                               contract( y1, delta, "relative" ),
+                               contract( y2, delta, "relative" )
+                               )
> inner
```

```
Object of class Intervals
1 interval over R:
[0.500000007450581, 0.749999988824129]
```

```
> outer <- interval_intersection(
+                                expand( y1, delta, "relative" ),
+                                expand( y2, delta, "relative" )
+                                )
> outer
```

```
Object of class Intervals
2 intervals over R:
[0.499999992549419, 0.750000011175871]
[0.999999985098839, 1.00000000745058]
```

Finally, we identify points which may or may not be in the intersection, depending on whether we make a conservative, literal, or anti-conservative interpretation of the nominal endpoints.

```
> interval_difference( outer, inner )
```

```
Object of class Intervals_full
3 intervals over R:
[0.499999992549419, 0.500000007450581)
(0.749999988824129, 0.750000011175871]
[0.999999985098839, 1.00000000745058]
```

The `expand` and `contract` methods have other uses as well. Here, we eliminate gaps of size 2 or smaller:

```
> x <- Intervals( c(1,10,100,8,50,200), type = "Z" )
> x
```

```
Object of class Intervals
3 intervals over Z:
[1, 8]
[10, 50]
[100, 200]
```

```
> w <- 2
> close_intervals( contract( reduce( expand(x, w/2) ), w/2 ) )
```

```
Object of class Intervals
2 intervals over Z:
[1, 50]
[100, 200]
```

# 4 Notes on implementation

## 4.1 Endpoint representation

For the moment, interval endpoints are always stored using R's *numeric* data type. Although this is wasteful from an memory and speed point of view, we do it for two reasons. First, use of R's `Inf` and `-Inf` — not possible with the *integer* type — is very convenient when computing complements. Second, the range of integers which can be represented using the *numeric* data type is considerably greater:

```
> .Machine$integer.max

[1] 2147483647

> numeric_max <- with( .Machine, double.base^double.digits )
> options( digits = ceiling( log10( numeric_max ) ) )
> numeric_max

[1] 9007199254740992
```

## 4.2 Efficiency

All computations are accomplished by treating intervals as pairs of tagged endpoints, sorting these endpoints (along with their tags), and then making a single pass through the results. Computational complexity for set operations is therefore $O(n \log n)$, where input object $i$ contains $n_i$ rows and $n = \sum_i n_i$. The same sorting approach is also used for `interval_overlap`, although if every interval in a query object of $m$ rows overlaps every intervals in a target object of $n$ rows, generating output alone must of necessity be $O(mn)$.

Sorted endpoint vectors are not retained in memory. If one wishes to query a particular object over and over, repeated sorting would be inefficient; in practice so far, however, such repeated querying has not been needed.

## 4.3 Checking validity

The code behind `interval_overlap` and `reduce` (key methods in the *intervals* package, which may be directly called by the user and are also used internally in numerous locations) is written in C++ for efficiency. The compiled code makes a number of assumptions about the `SEXP` objects passed in as arguments, but does not explicitly check these assumptions. Nonetheless, when the R wrappers for the compiled code are applied to *valid* objects from the `Intervals` or `Intervals_full` classes, all assumptions will always be met. This design decision was taken so that the requirements for individual objects and their contents could be gathered together in a single, natural location: the classes' `validity` functions.

The *intervals* package provides replacement methods — e.g., `type` and `closed` — which implement error checking and preserve object validity. R's implementation of S4 classes, however, leaves object data slots exposed to the user. As a consequence, a user can directly manipulate the data slots of a valid `Intervals` or `Intervals_full` object in a way that invalidates the object, but does not generate any warning or error.

To prevent invalid objects from being passed to compiled code — and potentially generating segmentation faults or other problems — all wrapper code in this package includes a `check_valid` argument. This argument is set to `TRUE` by default, so that `validObject` is called on relevant objects before handing them off to the compiled code. For efficiency, users may choose to override this extra check if they are certain they have not manually assigned inappropriate content to objects' data slots.

# 5 Session information

- R version 2.8.1 (2008-12-22), `i386-apple-darwin8.11.1`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: customizations 1.5.1, intervals 0.10.3