# Introduction to processdata

Niels Richard Hansen[1], *University of Copenhagen*

## Contents



## 1 Summary

The package `processdata` implements a hierarchy of S4 classes for storing data for discretely observed, multivariate stochastic processes.

The most important feature of the implemented classes is to provide an interface to data sets that contain components that are not easily or conveniently stored in a standard way, e.g. as a single matrix or data frame. In particular, joint data of event times (point process data) and discretely observed continuous processes is supported. Moreover, the data structures handle process data for multiple units (e.g. longitudinal data), and the joint representation of different types of data allows for a simple interface to subscription and subsetting that does not rely on knowledge about the actual implementation.

Why do we need yet another data structure in R? Because no data structure currently exists that can handle the data described without either excessive amounts of redundant information or a lot of bookkeeping. It is possible to implement the S4 classes in

[1]Postal address: Department of Mathematical Sciences, University of Copenhagen Universitetsparken 5, 2100 Copenhagen Ø, Denmark.
Email address: Niels.R.Hansen@math.ku.dk

processdata based on a single data frame representation of the data, but that representation will include redundant information, and for the intended applications this will often make it prohibitively large. The alternative is that we keep the data in several different data frames to avoid the redundancies, but this will impose an error prone responsibility for the user to maintain that the different parts of the data are always linked correctly together.

Using the classes ContinuousProcess and MarkedPointProcess in processdata we can, conceptually, view the data as a single big data frame while the implementation attempts to minimize redundancies. The classes are responsible for keeping track of the links between the internal representations, but for the end user this is irrelevant. A range of standard methods for subscription and subsetting is implemented for the classes, which, together with sensible default methods for plotting the data, provides the fundamental framework in R for handling data that combine discrete events with continuous process data.

# 2 Getting started

## 2.1 The first object

A bivariate Brownian motion can be represented in R as a $n \times 2$ matrix, e.g.

```
> n <- 2001
> p <- 2
> X <- apply(matrix(rnorm(n*p), n, p), 2, cumsum)
```

We can turn this matrix into an object of class ContinuousProcess by a call of the constructor continuousProcess.

```
> processX <- continuousProcess(X)
```

The resulting object can be printed, which only shows minimal structural information, and it can be summarized, which provides additional information – in the case of continuous variables the default is to compute the median per variable.

```
> print(processX)
```

```
Class: ContinuousProcess
Dimensions: 2001 x 2
Index variable: time

2 variables.
   V1 V2
```

```
> summary(processX)

Class: ContinuousProcess
Dimensions: 2001 x 2
Index variable: time

2 variables.
   V1 V2

  grid-points time-range median(V1) median(V2)
1        2001   [0;2000]  -16.06047   4.330141
```

The constructor makes a number of decisions, since the raw data set in `X` does not contain any information on variable names, time points of observations etc. In this case it automatically labels the variables `V1` and `V2`, and assumes that the observations are equidistant and observed at time points `0, 1, ..., 2000`. The summary tells us that the data set contains 2001 observations (the number of grid points in the observation grid) in the observation window from 0 to 2000.

We can instead provide the information for the constructor.

```
> X <- cbind(seq(0, 1, length.out = n), X)
> colnames(X) <- c("time", "Brownian", "Motion")
> processX <- continuousProcess(X)

> summary(processX)

Class: ContinuousProcess
Dimensions: 2001 x 2
Index variable: time

2 variables.
   Brownian Motion

  grid-points time-range median(Brownian) median(Motion)
1        2001      [0;1]        -16.06047       4.330141
```

The class handles an equidistant and a non-equidistant observation grid equally well.

## 2.2 Combining more information

Besides the formatting of the print and summary information, the `ContinuousProcess` object seems to be just a wrapped up version of the data matrix. This is, from a conceptual

point of view, correct. The more useful features of the class are first discovered when we have more complicated data sets.

We extend the data set to hold information of the sampling of *two* bivariate Brownian motions. To do this we introduce an `id` variable and change the data format for the "raw" data to a data frame.

```
> XX <- apply(matrix(rnorm(n*p), n, p), 2, cumsum)
> XX <- cbind(seq(0, 1, length.out = n), XX)
> X <- as.data.frame(rbind(X, XX))
> X <- cbind(data.frame(id = rep(c("A", "B"), each = n)), X)
> processX <- continuousProcess(X)
```

Summarizing the data set reveals the structure we now have.

```
> summary(processX)
```

```
Class: ContinuousProcess
Dimensions: 4002 x 2
Index variable: time

2 units.
   id: A B

2 variables.
   Brownian Motion

  grid-points time-range median(Brownian) median(Motion)
A        2001      [0;1]        -16.06047       4.330141
B        2001      [0;1]         -2.97718      14.940415
```

There are two *units* named `A` and `B`, respectively, and we get summary information per unit. Still, the new object is conceptually just a layer around the data frame that provides a print and a summary method. A real advantage is first gained when we, in addition to the observed processes, have unit specific data. The two units may be at different locations, say.

```
> unitData <- data.frame(id = c("A", "B"),
+                        location = c("town", "country"))
> processX <- continuousProcess(X, unitData = unitData)
```

```
> summary(processX)
```

```
Class: ContinuousProcess
Dimensions: 4002 x 3
Index variable: time

2 units.
   id: A B

3 variables.
   location Brownian Motion

  grid-points time-range location median(Brownian) median(Motion)
A        2001      [0;1]     town        -16.06047       4.330141
B        2001      [0;1]  country         -2.97718      14.940415
```

The additional, unit specific, information on location is now integrated into the data set. Conceptually it looks as if we added a location column to the data frame `X`, which contains 2001 repeated values of `town` and 2001 repeated values of `country`. However, the data structure does actually not make this wasteful copy of entries, which would be a real burden for large data sets. Instead it just maintains the relation between the process data and the unit data without the actual copying of data.

## 2.3 Including point process data

A primary motivation for implementing the `processdata` package was to combine the previously described data structures with time points of events. This is implemented by the S4 class `MarkedPointProcess`, which is an extension of the `ContinuousProcess` class.

Having constructed the `processX` data set already, we can easily augment the data set with event times.

```
> pointData <- data.frame(id = c("A", "A", "A", "B", "B"),
+                         time = c(0.2413, 0.7622, 0.9724, 0.1100, 0.6414))
> processX <- markedPointProcess(pointData, continuousData = processX)
> summary(processX)

Class: MarkedPointProcess
Dimensions: 4006 x 4
Index variable: time

2 units.
   id: A B

4 variables.
```

```
   location Brownian Motion point

  grid-points time-range location median(Brownian) median(Motion) #point
A        2004      [0;1]     town       -16.050565       4.316669      3
B        2002      [0;1]  country        -2.976267      14.955649      2
```

The summary shows that we have added a column with variable name `point`, that the dimensions have changed to $4006 \times 4$, and that the column providing the summary of points shows the number of points per unit – with 3 points observed for unit `A` and 2 points observed for unit `B`.

We can think of the data structure as appending a `point` column consisting of 0-1 values to the remaining data set, where 1 represents an event. However, to do that we need to modify the observation grid to include all time points where we have observed events. This is done by augmenting the observation grid with the time points of events *not already in the observation grid* and extrapolating the values of process data to the agumented grid points *from the left*. In the example above, there are five points, but only four new time points as 0.1100 is already in the observation grid.

As with the unit specific data for the `ContinuousProcess` data structure, the `MarkedPointProcess` data structure does not append a wasteful 0-1 column consisting mostly of 0's. The data structure is keeping track of the relations between the different types of data and allows us to work with the data *as if* the data set is a $4006 \times 4$ dimensional matrix of observations.

With equidistant observations for the original data set the inclusion of event times can result in a non-equidistant observation grid. This may or may not be desirable. It retains the precision of the event times, but for the down-stream analysis it may be desirable to coarsen these time points to the equidistant grid. This is done by the `coarsen` argument to the constructor. We do the construction of the process object here in one go from the three data frames.

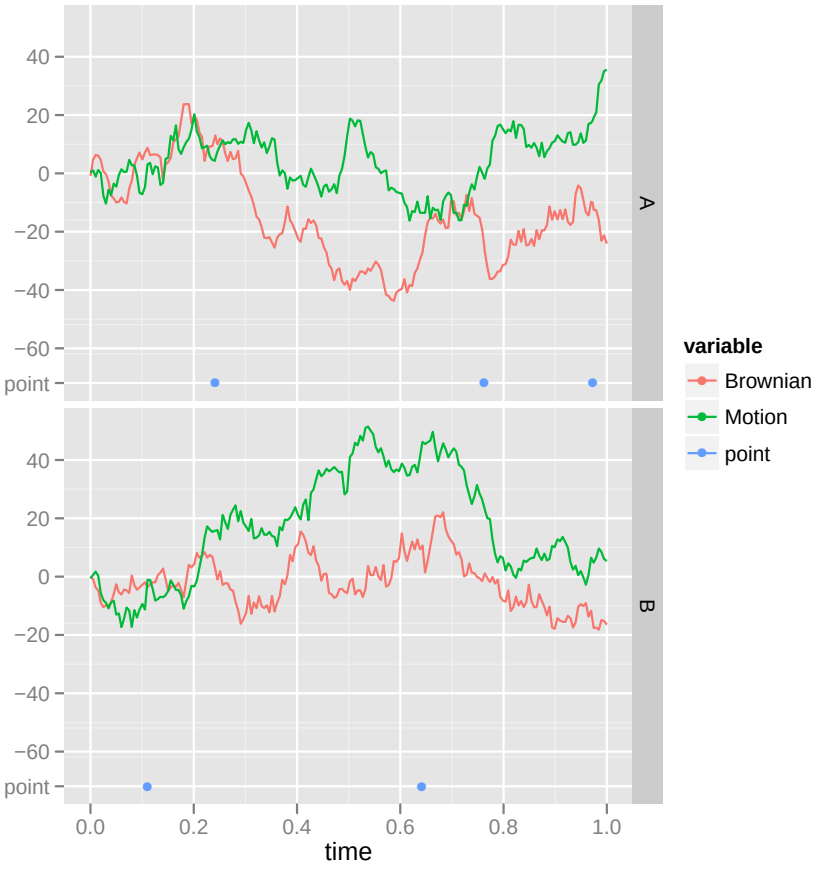
```
> processX <- markedPointProcess(pointData, continuousData = X,
+                                unitData = unitData, coarsen = 'right')
> summary(processX)

Class: MarkedPointProcess
Dimensions: 4002 x 4
Index variable: time

2 units.
   id: A B

4 variables.
   location Brownian Motion point
```

```
  grid-points time-range location median(Brownian) median(Motion) #point
A        2001      [0;1]     town         -16.06047       4.330141      3
B        2001      [0;1]  country          -2.97718      14.940415      2
```

# 3 Plotting

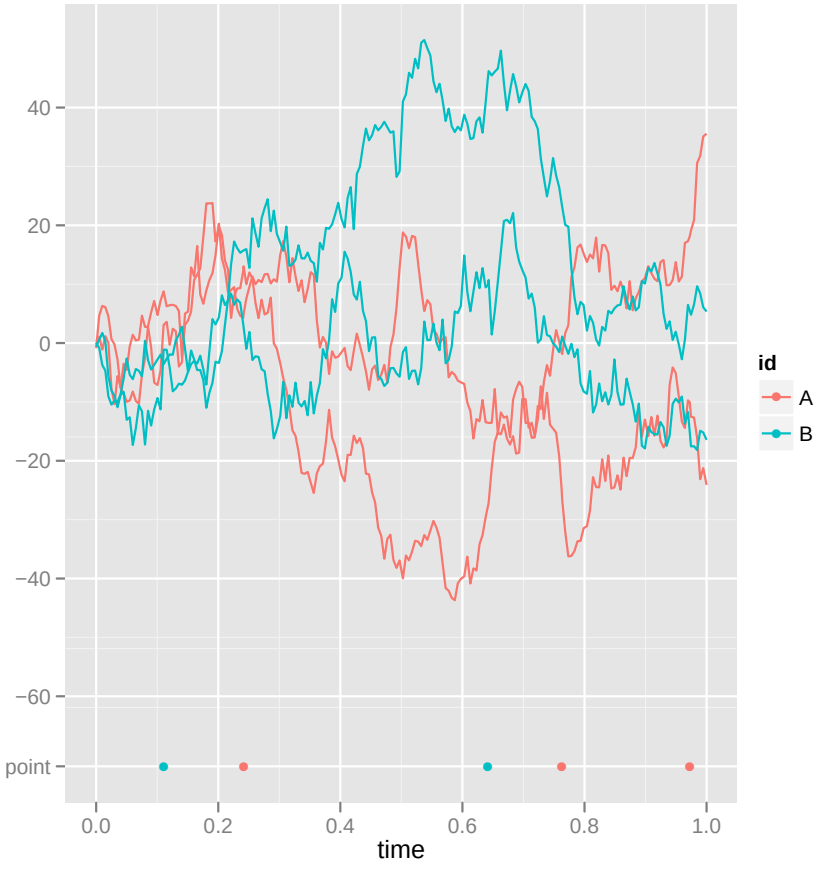
The package provides a plot method based on the `ggplot2` package.

```
> print(plot(processX))  ##print only needed for Sweave
```




A call of `plot` as above is all that is needed to plot the data in a sensible way. However, depending upon the structure and size of the data set, various modifications of the plot may be desirable. In particuar, we can modify the plot using the possibilities implemented in `ggplot2`, or we can plot a subset of the data only using the subsetting functionality described below.

```
> p <- plot(processX) + facet_null() + aes(colour = id)
> print(p) ##print only needed for Sweave
```




Using simple tools from `ggplot2` we can, as above, modify the way the data are plotted. Instead of different panels for each unit, we can colour code the units and drop the colour coding of the variables.

# 4 Subsetting

The standard square bracket, `[,]`, notation for subsetting can be used with `numeric`, `logical` and `character` vectors just as if the data set was a data frame. To this end, a few things are important to know. First, the unit id and the observation grid are *not* regarded as columns (variables) in the data set, but rather as the structure of the data set. Thus `processX` has 4 columns, whose names are the 4 variable names listed in the summary. Second, the rows do not have names, and character vector subsetting in the first coordinate means subsetting to a set of unit ids.

```
> processX[ ,c(1,2)]

Class: MarkedPointProcess
Dimensions: 4002 x 2
Index variable: time

2 units.
   id: A B

2 variables.
   location Brownian

> processX[1:3000, ]

Class: MarkedPointProcess
Dimensions: 3000 x 4
Index variable: time

2 units.
   id: A B

4 variables.
   location Brownian Motion point

> processX[-(3001:4000), c(1,4)]

Class: MarkedPointProcess
Dimensions: 3002 x 2
Index variable: time

2 units.
   id: A B

2 variables.
   location point

> summary(processX["A", c("location", "Brownian", "point")])

Class: MarkedPointProcess
Dimensions: 2001 x 3
Index variable: time
```

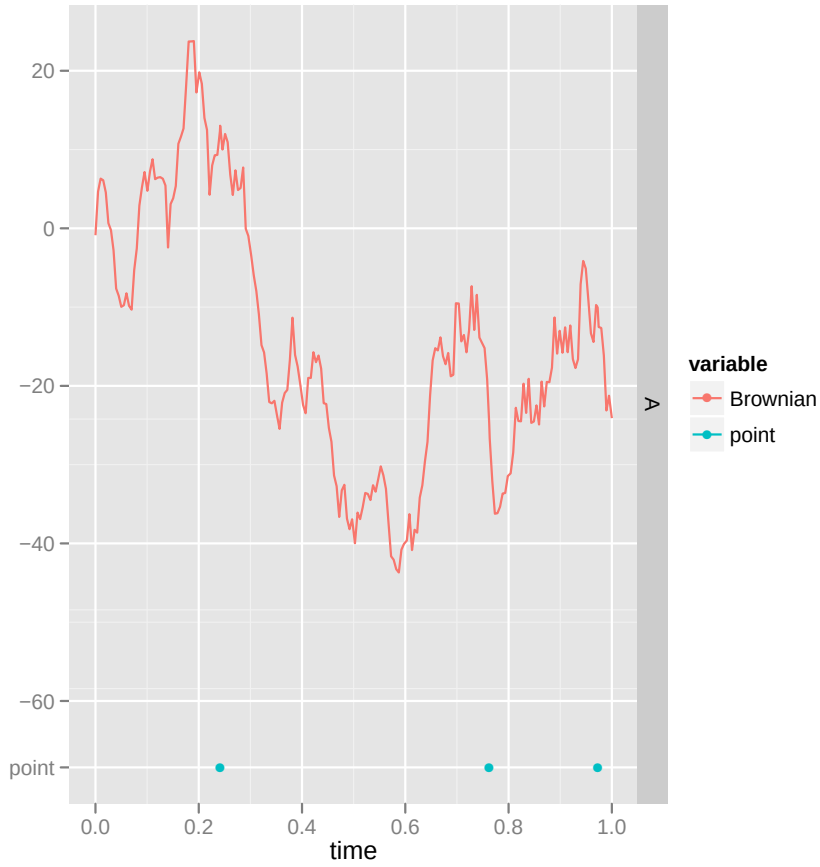
```
3 variables.
   location Brownian point

  grid-points time-range location median(Brownian) #point
A        2001      [0;1]     town        -16.06047      3
```

Bracket subsetting supports the `drop` argument, which by default is always `FALSE`. The consequence of "dropping" depends upon the context. In all cases subsetting to a single column with `drop = TRUE` returns a vector with the values of the column. For a `MarkedPointProcess` object and subsetting to 2 or more columns containing unit data or continuous process data with `drop = TRUE` results in a `ContinuousProcess` object.

```
> processX[, -4, drop = TRUE]


Class: ContinuousProcess
Dimensions: 4002 x 3
Index variable: time

2 units.
   id: A B

3 variables.
   location Brownian Motion


> head(processX[ , 2, drop = TRUE])


[1] -0.876766973 -1.623156401 -0.247241610 -0.009659645  0.098967827
[6]  1.301689103


> processX[ , 4, drop = TRUE]


[1] 0.2415 0.7625 0.9725 0.1105 0.6415
```

The combination of subsetting and plotting can be a useful for visual exploration of a larger data set.

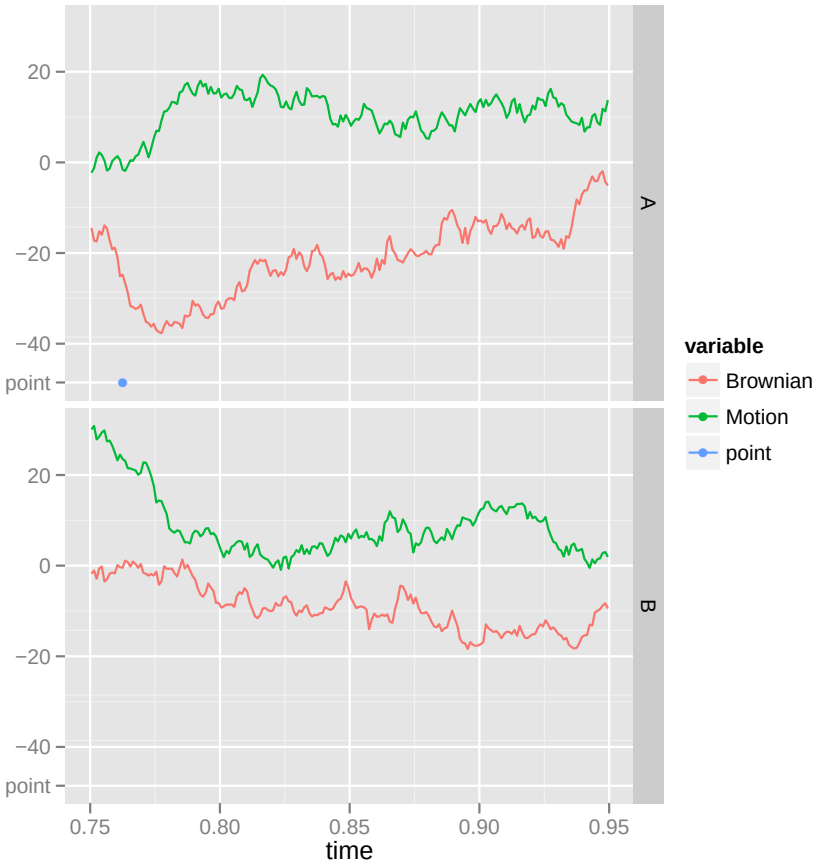
```
> print(plot(processX["A",c(2,4)]))  ##print only needed for Sweave
```




In addition to bracket subsetting there is a `subset` method, which extracts subsets of a data set via expressions that evaluate to logical. The expressions are *logical filters*, which can be specified in terms of column variables in the data set, the unit id (default `id`) variable and the index variable (default `time`).

Subsetting using the `time` variable in the logical filter results in a zooming feature for the plot method.

```
> ##print only needed for Sweave
> print(plot(subset(processX, time > 0.75 & time < 0.95)))
```




## 5 Summary of the interface

The sections above introduced the two constructors, `continuousProcess` and `markedPointProcess`, the bracket subsetting, the `subset` method and the `plot` method.

The `ContinuousProcess` class has, furthermore, several methods for extracting the data in the data set. The extractors `getTime` and `getId` extract the observation grid and the unit ids, respectively, as vectors. The method `getUnitData` extracts the unit specific data as a data frame.

For the `MarkedPointProcess` class there are additional extractor methods `getPointTime` and `getPointId` that extract the event times and corresponding event unit ids, respectively.

Both classes have the self-explaning `dim` and `colNames` methods.

A central method for data extraction is `getColumns`. It takes a character or numeric second argument (which can be a vector), and returns the selected data column(s). If two or more

columns are selected, a list is returned with the columns. If one column is selected, the return value is a vector by default.

In addition there are convenience extractors, `getFactors` and `getNumerics` that extract the factor and numeric columns, respectively, for the continuous process data, together with `getMarkType` and `getMarkValue` that extract marks and mark values, respectively, for the event times. See their help pages for details on their return values.

# 6 Technical details

The actual implementation is in principle irrelevant to the end user. Only the constructors and the interface to access and extract the data, as summarized above, are needed. This section gives a some more technical details for the interested on the implementation and the internal design.

First, the actual data are regarded as *immutable*. The intention was to design a package where the different data elements can be initially combined and then later accessed and used, and for which sub data sets can be easily selected. The decision was to store the initial data as immutable data objects and then provide extration and subsetting functionality. It was never the intention that the data format should be suitable for altering entries in the data set, e.g. in relation to data clearning.

The current implementation relies on an internal representation of the data that are stored in environments pointed to by slots in the objects. A practical consequence, important for working with large data sets, is that the actual data are not copied around if copies of the data set are made. Even subsetting does not require copying the data as these operations rely on pointers to the subset of the full data set instead of actually copying the subset of the data. This also allows for an `unsubset` function that resets all pointers and restores the data set to the form of the orginal data set.

We can investigate the structure of the objects with the two methods `object.size` and `str`.

```
> str(processX)
```

```
Formal class 'MarkedPointProcess' [package "processdata"] with 18 slots
  ..@ markColNames      : chr "point"
  ..@ markValueColNames : chr(0)
  ..@ iPointSubset      : int -1
  ..@ jPointSubset      : int -1
  ..@ pointPointer      : int -1
  ..@ factorColNames    : chr(0)
  ..@ numericColNames   : chr [1:2] "Brownian" "Motion"
  ..@ iSubset           : int -1
```

```
  ..@ jSubset           : int -1
  ..@ positionVar       : chr "time"
  ..@ equiDistance      : num 0
  ..@ metaData          : list()
  ..@ unitColNames      : chr "location"
  ..@ idVar             : chr "id"
  ..@ iUnitSubset       : int -1
  ..@ jUnitSubset       : int -1
  ..@ pointProcessEnv [environment] with 3 variables
    ..$ id           : Factor w/ 2 levels "A","B": 1 1 1 2 2
    ..$ markType     : Factor w/ 1 level "point": 1 1 1 1 1
    ..$ pointPointer : int [1:5] 484 1526 1946 2223 3285
  ..@ valueEnv [environment] with 5 variables
    ..$ id           : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
    ..$ position     : num [1:4002] 0 0.0005 0.001 0.0015 0.002 0.0025 0.003 0.0035 0.004
    ..$ unitData     :'data.frame':        2 obs. of  1 variable:
      ..$ location: Factor w/ 2 levels "country","town": 2 1
    ..$ var_Brownian : num [1:4002] -0.87677 -1.62316 -0.24724 -0.00966 0.09897 ...
    ..$ var_Motion   : num [1:4002] 0.285 1.173 0.276 -0.3 -0.358 ...
```

```
> object.size(processX)
```

```
114956 bytes
```

The `str` method is a slightly modified version of the default method, which digs into the environments and lists the structures of their content. Likewise, `object.size` is a slightly modified version of the default method that sums up the sizes of the slots as well as the content of the environments.

A slightly odd consequence of the fact that the actual data are stored in environments is that any subset points to the environments with the complete data. Hence, in addition to the pointers in the subset, the sum of the sizes of objects in the environments and the slots is *larger* for a subset than for the original data. In isolatation this is regarded as the relevant way to measure the size of a subset, and if we, for instance, save the object using `save` then everything in the environments is saved. On the other hand, with several objects all being subsets of one data set, the actual memory consumption is (much) smaller than the sum of the sizes of the objects as measured by `object.size`.

```
> processXsubset <- processX[1:1000, c(2,4)]
> object.size(processXsubset)
```

```
118940 bytes
```

The constructors `continuousProcess` and `markedPointProcess` can be called with a `ContinuousProcess` and a `MarkedPointProcess` object, respectively, which will generate a true copy of the object. In particular, if we have a subset of the data set on which we call the constructor, we create a true copy of the object with the subset of the data now stored in a new environment.

```
> processXtruesubset <- markedPointProcess(processXsubset)

Class: MarkedPointProcess
Dimensions: 1000 x 2
Index variable: time

2 variables.
   Brownian point

> object.size(processXtruesubset)

22372 bytes

> identical(unsubset(processXtruesubset), processXtruesubset)

[1] TRUE

> identical(unsubset(processXsubset), processX)

[1] TRUE
```

The current implementation is intended to be useful for working with medium sized data sets. Some effort has been put into avoiding the storage of redundant information while integrating event data and process data. Moreover, the avoidance of data copying may be useful if we want to fit many statistical models, and each model needs to carry a copy of the data, or if we want to fit the same model repeatedly to bootstrapped or randomly subsetted data. However, the data representation still requires that the whole data set fits into the physical memory. On the other hand, by providing an interface to marked point process data we can write an extension, `BigMarkedPointProcess`, say, of the `MarkedPointProcess` class that either rely on even more efficient internal storage of the data or access to the data in a data base or in another disk based storage format.