An introduction to analysing disease outbreak data using *epibase* 0.1-0

Thibaut Jombart, Addyour Namehere

April 24, 2013

Abstract

This vignette introduces the main functionalities of *epibase*, a package implementing basic tools for the analysis of outbreak data. Disease outbreak data can be varied and complex, and one of the core features of *epibase* lies in the formal (S4) class `obkData`, which allows for storing and handling a variety of data about individuals, samples, contact structures, or even clinical events. Beyond introducing this data structure, this tutorial illustrates how these objects can be handled and analyzed in R.

Contents

1 Storing outbreak data

In this section, we first detail the structure of the core classes used for storing information in *epibase*, and then explain how to import data into the package, and how to handle data once imported.

1.1 Class definitions

Data collected during outbreaks can be hugely diverse and complex. In *epibase*, our purpose is to have a general class of objects which can store virtually any information sampled during outbreak, without the user worrying about storage issues. For most purposes, the core class `obkData` will be treated as a black box, with which the user can interact using specific functions called *accessors*, without having to worry about the internal structure of the object.

1.1.1 `obkData`: storage of outbreak data

The class `obkData` is used to store various types of information. The definition of the class in terms of R objects can be obtained by:

```
> library(epibase)
> getClassDef("obkData")


Class "obkData" [package "epibase"]

Slots:

Name:         individuals             samples            clinical
Class:    dataframeOrNULL    dataframeOrNULL         listOrNULL

Name:                 dna            contacts              trees
Class: obkSequencesOrNULL  obkContactsOrNULL   multiPhyloOrNULL
```

One can also examine a structure using an empty object:

```
> new("obkData")


=== obkData object ===
== Empty slots ==
 @individuals, @samples, @clinical, @dna, @contacts, @trees
```

Each slot of an `obkData` object is optional. By convention, empty slots are always `NULL`. The slots respectively contain:

- `@individuals`: a `data.frame` storing individual data, such as age, sex, or onset of symptoms. If not `NULL`, this `data.frame` will have exactly one row per individual, with row names providing unique identifiers for individuals.
- `@samples`: a `data.frame` storing sample data, typically swab results or accession numbers of DNA sequences. If not `NULL`, this `data.frame` must contain the three following columns: `individualID` (unique identifiers for individuals), `sampleID` (unique identifiers for samples), and `date` (collection dates for the samples).

- `@clinical`: a list of `data.frames` storing any additional clinical information; there is no constraint on the number of `data.frames` stored, but each one must contain columns named `individualID` (unique identifiers for individuals) and `date` (date of observations/interventions).
- `@dna`: DNA sequences of one or more genes, stored as an `obkSequences` object. See section below for details on `obkSequences` objects.
- `@contacts`: dynamic contact network between the individuals, stored as an `obkContacts` object. See section below for details on `obkContacts` objects.
- `@trees`: a list of phylogenetic trees with the class `multiPhylo` (from the *ape* package); can be used to store posterior distribution of trees from Bayesian software such as BEAST.

The slots of an object `foo` can be accessed using `foo@[name-of-the-slot]`. Let us use a toy dataset created taken from the `obkData` documentation; for now, we overlook the data creation process, and just focus on the content of the object `x`:

```
> class(x)


[1] "obkData"
attr(,"package")
[1] "epibase"


> slotNames(x)


[1] "individuals" "samples"     "clinical"    "dna"         "contacts"
[6] "trees"


> x


=== obkData object ===
== @individuals==
         age sex
toto      20   m
John Doe  18   m
titi      67   ?

== @samples==
  individualID sampleID       date swab sequenceID
1         toto        1 2001-02-13    +      No304
2         toto        3 2001-03-01    -      No306
3         titi        2 2001-05-25    +      No305

== @dna==
[ 3 DNA sequences in 1 locus ]

[[1]]
3 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No306 No304
```

```
Base composition:
    a     c     g     t
0.306 0.260 0.126 0.307

== Empty slots ==
 @clinical, @contacts, @trees
```

`x` is an `obkData` object containing information on individuals (three people, with unique identifier for each row), samples (the same sample could be appearing multiple times), and some DNA sequences. Note the presence of the mandatory columns in `@samples`: `individualID`, `sampleID`, and `date`. As DNA sequences are also present, `@samples` also contains a column `sequenceID` with unique sequence identifier, matching labels used in `@dna`. As no locus information is provided in `@samples`, it is assumed that all sequences are of the same gene. Otherwise, there should be a column `locus` in `@samples` containing this information. Accessing a given slot is as easy as:

```
> x@individuals
```

```
         age sex
toto      20   m
John Doe  18   m
titi      67   ?
```

```
> x@samples
```

```
  individualID sampleID       date swab sequenceID
1         toto        1 2001-02-13    +      No304
2         toto        3 2001-03-01    -      No306
3         titi        2 2001-05-25    +      No305
```

1.1.2 `obkSequences`: storage of DNA sequences for different genes

Pathogen sequence data can typically be obtained for a range of different genes, making the handling of such information not entirely trivial. The class `obkSequences` stores such information. It consists in a list of matrices of aligned DNA sequences (in rows), stored using *ape*'s class `DNAbin` for efficiency, with each item of the list corresponding to a gene. If provided, gene names are the names of the list. The row names for each matrix contain unique identifiers for the sequences, typically accession numbers. Matching of samples and sequences made in the `obkData` objects through the field `sequenceID` in the `@sample` slot. When several loci have been sequenced, the locus information must also be provided for each accession number using a column `locus` in the `@sample` slot.

Again, let us look at a toy example without paying attention to how data are created:

```
> class(x)
```

```
[1] "obkSequences"
attr(,"package")
[1] "epibase"
```

```
> slotNames(x)
```

```
[1] "dna"
```

```
> x
```

```
[ 15 DNA sequences in 3 loci ]

$gene1
10 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No304 No306 No0906S No0908S No0909S ...

Base composition:
    a     c     g     t
0.307 0.261 0.125 0.307

$HighGene
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No1007S

Base composition:
    a     c     g     t
0.306 0.265 0.126 0.303

$`Phage foobar`
4 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No1114S No1202S No1206S No1208S

Base composition:
    a     c     g     t
0.305 0.262 0.128 0.306
```

```
> class(x@dna$"gene1")
```

```
[1] "DNAbin"
```

`x` is an `obkSequences` object containing three genes. Data are stored in the unique slot `@dna`, which is a list of `DNAbin` matrices.

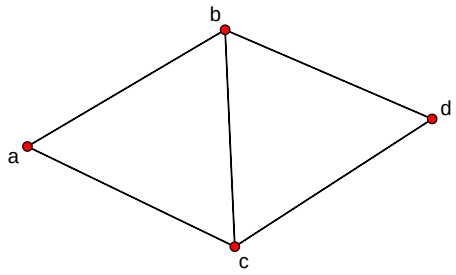
1.1.3 `obkContacts`: storage of dynamics contact networks

obkData objects can also store contact data between individuals, in the slot `@contacts`. These contacts can be fixed or vary in time, in which case data are stored as a dynamic contact network. The slot `@contacts` is an instance of the class `obkContacts`, which currently contains either a `network` object (static graph, from the *network* package), or a `networkDynamic` object, for contacts varying in time (from the *networkDynamic* package). These objects are fully documented in their respective vignettes. Here, we detail a simple toy example from the documentation of `obkContacts`:

```
> cf <- c("a", "b", "a", "c", "d")
> ct <- c("b", "c", "c", "d", "b")
> onset <- c(1, 2, 3, 4, 5)
> terminus <- c(1.2, 4, 3.5, 4.1, 6)
> oc.static <- new("obkContacts",cf,ct,FALSE) # static network
> oc.dynamic <- new("obkContacts",cf,ct,FALSE,onset,terminus)
> oc.static


 Number of individuals = 4
 Number of contacts = 5
 Contacts = fixed
 Network attributes:
  vertices = 4
  directed = FALSE
  hyper = FALSE
  loops = FALSE
  multiple = TRUE
  bipartite = FALSE
  total edges= 5
    missing edges= 0
    non-missing edges= 5

 Vertex attribute names:
    vertex.names


> oc.dynamic


 Number of individuals = 4
 Number of contacts = 5
 Contacts = dynamic
 Network attributes:
  vertices = 4
  directed = FALSE
  hyper = FALSE
  loops = FALSE
  multiple = TRUE
  bipartite = FALSE
  total edges= 5
    missing edges= 0
    non-missing edges= 5

 Vertex attribute names:
    vertex.names
```

`oc.static` is a static, non-directed contact network (class `network`). It can be plotted easily using:

```
> plot(oc.static@contacts, displaylabels=TRUE, main="Static contact network")
```



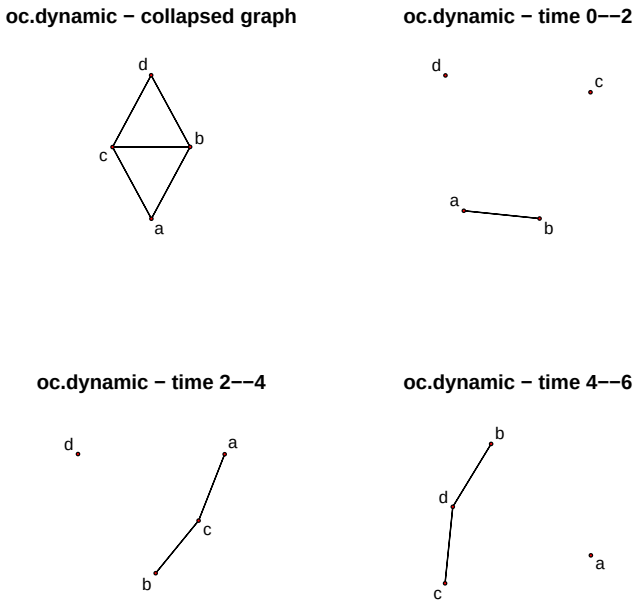
`oc.dynamic` is a dynamic graph, i.e. a graph whose vertices and edges can change over time. By default, plotting the object collapses the graph so that all vertices and edges that exist at some point are displayed; however, sections of the graph for given time intervals can be obtained using `network.extract`. As a reminder, here is the input of the graph `oc.dynamic`:

```
> data.frame(onset,terminus,ct,cf)
```

```
  onset terminus ct cf
1     1      1.2  b  a
2     2      4.0  c  b
3     3      3.5  c  a
4     4      4.1  d  c
5     5      6.0  b  d
```

And here are various plots, first of the full (collapsed) contact network, then for different time intervals (0–2, 2–4, 4–6):

```
> par(mfrow=c(2,2))
> plot(oc.dynamic@contacts,main="oc.dynamic - collapsed graph",
+      displaylabels=TRUE)
> plot(network.extract(oc.dynamic@contacts, onset=0, terminus=2),
+      main="oc.dynamic - time 0--2", displaylabels=TRUE)
> plot(network.extract(oc.dynamic@contacts, onset=2, terminus=4),
+      main="oc.dynamic - time 2--4", displaylabels=TRUE)
> plot(network.extract(oc.dynamic@contacts, onset=4, terminus=6),
+      main="oc.dynamic - time 4--6", displaylabels=TRUE)
```



networkDynamic allows for extensive manipulation of dynamic networks. For more information, refer to the vignette distributed with the package (`vignette("networkDynamic")`).

1.2 Getting data into *epibase*

Storing data in *epibase* requires the following, fairly simple steps:

1. read data into R
   (a) read `data.frame`s storing individuals, samples, and clinical information in R from a text file, typically using `read.table` or `read.csv` for comma-separated files. Every standard spreadsheet software can export data to these formats.
   (b) read DNA sequences from a single file, typically using `read.dna` from the ape package; this "master" file must contain all DNA sequences of all genes, with unique identifiers for the sequences as labels
2. use this information as input to the `obkData` constructor (`new("obkData",...)`) to create an `obkData` object.

In the following, we assume that step 1 is sorted and focus on step 2: using the constructor.

1.2.1 The `obkData` constructor

New objects are created using `new`, with these slots as arguments. If no argument is provided, an empty object is created, as seen before:

```
> new("obkData")


=== obkData object ===
== Empty slots ==
 @individuals, @samples, @clinical, @dna, @contacts, @trees
```

This function accepts the following arguments, which mirror to some extent the structure of the object (see `?obkData` for more information):

- `individuals`: a `data.frame` with a mandatory column named 'individualID', providing unique identifiers for the individuals.
- `samples`: a `data.frame` with 3 mandatory columns named 'individualID', 'sampleID', and 'date', providing identifiers for the individuals, for the samples, and dates. Dates must be provided in a way convertible to `Date` (see `?as.Date`). Default format for dates provided as characters is "%Y-%m-%d" (e.g. 1984-09-23). Alternative format can be specified via the argument `date.format`.
- `clinical`: a list of `data.frame`s, each of which has 2 mandatory fields, individualID' and 'date' (specified as before).
- `dna`: a list of DNA sequences in `DNAbin` or `character` format, as read by `read.dna`.
- `contacts`:
- `contacts`: a matrix of characters indicating edges using two columns; if contacts are directed, the first column is 'from', the second is 'to'; values should match individual IDs (as returned by get.individuals(x)); if numeric values are provided, these are converted to integers and assumed to correspond to individuals returned by get.individuals(x).
- `contacts.start`: a vector of dates indicating the beginning of each contact.
- `contacts.end`: a vector of dates indicating the end of each contact.
- `contacts.duration`: another way to specify contacts.end, as duration of contact.
- `contacts.directed`: a logical indicating if contacts are directed; defaults to FALSE.
- `trees`: a list of phylogenetic trees in the class `multiPhylo` (from the ape package); this is basically a list of `phylo` objects, with the class "`multiPhylo`".

We can now show how the toy example previously used was created. Arguments `ind` and `samp` are `data.frame`s with some of the required fields:

```
> ind
```

```
  individualID age sex
1         toto  20   m
2     John Doe  18   m
3         titi  67   ?
```

```
> samp
```

```
  individualID sampleID       date swab sequenceID
1         toto        1 2001-02-13    +      No304
2         toto        3 2001-03-01    -      No306
3         titi        2 2001-05-25    +      No305
```

```
> samp$date
```

```
[1] 2001-02-13 2001-03-01 2001-05-25
Levels: 2001-02-13 2001-03-01 2001-05-25
```

Note that dates are in the right format, but are actually mis-specified as they are stored as a `factor`. As this is frequent (characters are stored as factors in `data.frame`s unless `options("stringsAsFactors")` is set to `FALSE`), the constructor is designed to accommodate this issue.

DNA sequences have been taken from `data(woodmouse)` for the sake of merely have sequences to play with:

```
> data(woodmouse)
> dat.dna <- as.list(woodmouse)
> dat.dna
```

```
15 DNA sequences in binary format stored in a list.

All sequences of same length: 965

Labels: No305 No304 No306 No0906S No0908S No0909S ...

Base composition:
    a     c     g     t
0.307 0.261 0.126 0.306
```

And then `obkData` object was simply created using:

```
> x <- new("obkData", individuals=ind, samples=samp, dna=dat.dna)
```

Note some slight differences from the inputs. Individuals labels are now used to name the rows of `@individuals`:

```
> x@individuals
```

```
         age sex
toto      20   m
John Doe  18   m
titi      67   ?
```

And dates are stored as proper dates, supporting basic mathematical operations:

```
> class(x@samples$date)

[1] "Date"

> x@samples$date

[1] "2001-02-13" "2001-03-01" "2001-05-25"

> x@samples$date + 1

[1] "2001-02-14" "2001-03-02" "2001-05-26"

> x@samples$date + 365

[1] "2002-02-13" "2002-03-01" "2002-05-25"
```

Some other, invisible checks have also been made when creating the object. For instance, it has been checked that every sampled individual is documented in `@individuals` (otherwise, a warning would have been issued), and that every sequence referred in `x@samples$sequenceID` was indeed in the list of DNA sequences in `@dna` (an error would have been issued otherwise, along with a list of faulty accession numbers).

1.2.2 The `obkSequences` constructor

In most cases, one will not need to construct `obkSequences` directly, this task being done implicitly will creating `obkData` objects. However, one might want to modify the DNA sequences stored in an existing `obkData`, thus needing to build a new `obkSequences`. As for `obkData`, `obkSequences` objects can be created using the constructor `new("obkSequences", ...)`, where "..." can be the following arguments:

- `dna`: a list of DNA sequences (not necessarily from the same gene) in `DNAbin` or `character` format; matrices will be accepted too if only one locus is provided. Sequences must be named using unique identifiers, typically accession numbers. Typically, this information will be obtained by reading sequence data into R using *ape*'s `read.dna` function.
- `locus`: an optional vector indicating the locus of each sequences; its length must match that of the list of sequences.

Using these inputs, the `obkSequences` constructor will sort out sequences per gene and store them as matrices, using one matrix per gene and checking that sequences from the same gene are actually of the same length.

Here, we illustrate the creation of `obkSequences` objects using a dataset of influenza sequences from *adegenet*, first read using `read.dna`:

```
> path.file <- system.file("files/usflu.fasta",package="adegenet")
> path.file


[1] "/Library/Frameworks/R.framework/Versions/2.15/Resources/library/adegenet/files/usflu.fasta"


> flu <- read.dna(path.file, format="fasta")
> flu


80 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY013200 CY013781 CY012128 CY013613 CY012160 CY012272 ...

Base composition:
    a     c     g     t
0.335 0.200 0.225 0.239
```

The object is simply created using:

```
> x <- new("obkSequences", dna=flu)
> x


[ 80 DNA sequences in 1 locus ]

[[1]]
80 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY013200 CY013781 CY012128 CY013613 CY012160 CY012272 ...

Base composition:
    a     c     g     t
0.335 0.200 0.225 0.239
```

As locus information is not provided, the constructor assumed (rightfully so) that all sequences are from the same (unnamed) locus. Here, the sequenced segment is actually hemagglutinin (HA), so we can add this information:

```
> x <- new("obkSequences", dna=flu, locus=rep("HA",80))
> x


[ 80 DNA sequences in 1 locus ]

$HA
80 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY013200 CY013781 CY012128 CY013613 CY012160 CY012272 ...

Base composition:
    a     c     g     t
0.335 0.200 0.225 0.239
```

Now, if we assume that for instance, the first 70 sequences were HA, followed by 8 neuraminidase (NA) and 2 nucleoprotein (NP), then we would use:

```
> x <- new("obkSequences", dna=flu, locus=rep(c("HA","NA","NP"), c(70,8,2)))
> x
```

```
[ 80 DNA sequences in 3 loci ]

$HA
70 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY013200 CY013781 CY012128 CY013613 CY012160 CY012272 ...

Base composition:
    a     c     g     t
0.335 0.200 0.226 0.239

$`NA`
8 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: EU199369 EU199254 CY031555 EU516036 EU516212 FJ549055 ...

Base composition:
    a     c     g     t
0.339 0.200 0.221 0.240

$NP
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY035190 EU852005

Base composition:
    a     c     g     t
0.339 0.200 0.220 0.241
```

Note that sequences do not have to be ordered by locus; the only thing that matters is that the argument `locus` matches the sequences provided in `dna`.

Replacing the `@dna` slot of an `obkData` object is as simple as:

```
> obj <- new("obkData")
> obj@dna


NULL


> obj@dna <- x
> obj


=== obkData object ===
== @dna==
[ 80 DNA sequences in 3 loci ]

$HA
70 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY013200 CY013781 CY012128 CY013613 CY012160 CY012272 ...

Base composition:
    a     c     g     t
0.335 0.200 0.226 0.239

$`NA`
8 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: EU199369 EU199254 CY031555 EU516036 EU516212 FJ549055 ...
```

```
Base composition:
    a     c     g     t
0.339 0.200 0.221 0.240

$NP
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1701

Labels: CY035190 EU852005

Base composition:
    a     c     g     t
0.339 0.200 0.220 0.241


== Empty slots ==
 @individuals, @samples, @clinical, @contacts, @trees
```

Note however, that this operation does not ensure matching of sequences IDs in `@dna` with the information provided in `@sample`.

1.2.3 The `obkContacts` constructor

2 Data handling using `obkData` objects

2.1 Accessors

The phylosophy underlying formal (S4) classes is that the internal representation of the data can be complex as long as accessing the information is simple. This is made possible by decoupling storage and accession: the user is not meant to access the content of the object directly, but has to use *accessors* to retrieve the information. In this section, we detail the existing accessors for object classes implemented in *epibase*. We use the notation "`[possible-values]`" to list or describe possible values of an argument; the symbols "`[]`" should be omitted from the actual command line. For instance:

```
myFunction(x, y=["foo" or "bar"])
```

means that the argument `y` of function `myFunction` can be either `"foo"` or `"bar"`, and proper calls would be:

```
> myFunction(x, y="foo")
```

or:

```
> myFunction(x, y="bar")
```

2.1.1 Accessors for `obkData` objects

Available accessors are also documented in `?obkData`. These functions are meant to retrieve information that is not trivially accessible. To simply access slots, use the `@` operator, e.g. `x@samples`, `x@individuals`, etc.

All accessors return `NULL` when information is missing, except for functions returning number of items, which will return `0`. In the following, we illustrate accessors using the toy dataset `x` generated by running:

```
> example(obkData)


> x


=== obkData object ===
== @individuals==
          age sex
toto       20   m
John Doe   18   m
titi       67   ?

== @samples==
    individualID sampleID       date swab sequenceID locus
1           toto        1 2001-02-13    +      No305 gene1
1.1         toto        1 2001-02-13    +      No304 gene1
2           toto        3 2001-03-01    -      No306 gene1
2.1         toto        3 2001-03-01    -    No0906S gene2
2.2         toto        3 2001-03-01    -    No0908S gene1
3           titi        2 2001-05-25    +    No0909S gene2

== @dna==
[ 6 DNA sequences in 2 loci ]

$gene1
4 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No304 No306 No0908S

Base composition:
    a     c     g     t
0.306 0.260 0.126 0.307

$gene2
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0906S No0909S

Base composition:
    a     c     g     t
0.309 0.261 0.124 0.306


== Empty slots ==
 @clinical, @contacts, @trees
```

- `get.individuals(x, data=["samples" or "individuals"])`: returns the individual IDs in `@samples` (default) or in `individuals`.
- `get.nindividuals(x, data=["samples" or "individuals"])`: returns the number of individuals in `@samples` (default) or in `individuals`.

```
> get.individuals(x)


[1] "toto"     "titi"     "John Doe"


> get.individuals(x, data="indiv")


[1] "toto"     "John Doe" "titi"
```

There are three individuals documented in individual data (`@individuals`), but samples for only two of them.

- `get.samples(x)`: returns the unique IDs of the samples in the data.

```
> get.samples(x)

[1] "1" "3" "2"
```

- `get.nsamples(x)`: returns the number of sample.

```
> get.nsamples(x)

[1] 3
```

- `get.locus(x)`: returns the names of the loci in the data.

```
> get.locus(x)

[1] "gene1" "gene2"
```

- `get.nlocus(x)`: returns the number of loci.

```
> get.nlocus(x)

[1] 2
```

- `get.sequences(x)`: returns the IDs of the sequences in `@dna`.

```
> get.sequences(x)

   gene11    gene12    gene13    gene14    gene21    gene22
 "No305"   "No304"   "No306" "No0908S" "No0906S" "No0909S"
```

- `get.nsequences(x)`: returns the number of sequences in `@dna`.

```
> get.nsequences(x)

[1] 6
```

- `get.trees(x)`: returns the content of `x@trees`.

```
> get.trees(x)

NULL
```

There is no tree in this object.

- `get.dna(x, locus=[`*locus IDs*`], id=[`*sequence IDs*`])`: returns a list of matrices of DNA sequences; the arguments `locus` and `id` are optional; if provided, they should be character strings corresponding to the name of the loci and/or sequences to be retained. Integers or logical will be treated as indicators based on the results of `get.locus` or `get.sequences`.

```
> get.dna(x)


$gene1
4 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No304 No306 No0908S

Base composition:
    a     c     g     t
0.306 0.260 0.126 0.307

$gene2
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0906S No0909S

Base composition:
    a     c     g     t
0.309 0.261 0.124 0.306
```

returns all the DNA sequences, in two matrices corresponding to different genes. We can request e.g. only the second gene:

```
> get.dna(x, locus=2)


$gene2
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0906S No0909S

Base composition:
    a     c     g     t
0.309 0.261 0.124 0.306
```

or even just specific sequences, say ("No305" and "No0906S"):

```
> get.dna(x, id=c("No305","No0909S"))


$gene1
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305

Base composition:
    a     c     g     t
0.304 0.262 0.129 0.306

$gene2
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0909S

Base composition:
    a     c     g     t
0.306 0.265 0.126 0.303
```

Note that we could also refer to sequences by their index in `get.sequences`:

```
> get.sequences(x)

   gene11    gene12    gene13    gene14    gene21    gene22
 "No305"   "No304"   "No306" "No0908S" "No0906S" "No0909S"

> identical(get.dna(x, id=c(1,6)), get.dna(x, id=c("No305","No0909S")))

[1] TRUE
```

- `get.data(x, data=[`*name of data seeked*`], where=NULL, drop=[`*TRUE/FALSE*`])`: multi-purpose accessor seeking a data field with a given name in the entire dataset; `data` can be the name of a slot, or the name of a column in `x@individuals`, `x@samples`, or `x@clinical`. The optional argument `where` allows one to specify in which slot the information should be looked for. The argument `drop` states whether to return a vector (`TRUE`), or a one-column `data.frame` (`FALSE`).

  For instance, we can retrieve swab results using:

```
> get.data(x,"swab")

[1] + + - - - +
Levels: - +
```

  or the sex of the different individuals:

```
> get.data(x, "sex", drop=FALSE)

         sex
toto       m
John Doe   m
titi       ?
```

  But searching for "sugarman" will return NULL as there is no such field in the data:

```
> get.data(x, "sugarman")

NULL
```

  And the same happens when looking for information in an empty slot:

```
> get.data(x, "date", where="clinical")

NULL
```

2.1.2 Accessors for `obkSequences` objects

Accessors of `obkSequences` objects are basically a subset of what is available for `obkData`. They work in the same way, and use the same arguments; they include:

- `get.locus`
- `get.nlocus`
- `get.sequences`
- `get.nsequences`
- `get.dna`

2.1.3 Accessors for `obkContacts` objects

2.2 Subsetting the data

A lot of data handling lies in creating subsets of the data based on some given criteria. The method **`subset`** for `obkData` objects allows for a range of manipulations. The syntax is as follows:

```
> subset(x, individuals=NULL, samples=NULL, locus=NULL, sequences=NULL,
+        date.from=NULL, date.to=NULL, date.format=NULL,
+        row.individuals=NULL, row.samples=NULL,...)
```

See `?subset.obkData` for the details of these arguments. The function works in a fairly intuitive way. The arguments **`individuals`**, **`samples`**, **`locus`** and **`sequences`** are vectors of characters indicating items to be kept. If integers or logicals are provided, these are assumed to match the output of `get.[...]`. For instance:

```
> get.individuals(x)


[1] "toto"     "titi"     "John Doe"


> subset(x, individual="titi")


=== obkData object ===
== @individuals==
     age sex
titi  67   ?

== @samples==
  individualID sampleID       date swab sequenceID locus
3         titi        2 2001-05-25    +    No0909S gene2

== @dna==
[ 1 DNA sequence in 1 locus ]

$gene2
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0909S
```

```
Base composition:
    a     c     g     t
0.306 0.265 0.126 0.303


== Empty slots ==
 @clinical, @contacts, @trees


> identical(subset(x, ind="titi"),subset(x, ind=2))


[1] TRUE


> identical(subset(x, ind="titi"),subset(x, ind=c(FALSE,TRUE)))


[1] TRUE
```

Another, non-exclusive way of subsetting the data is using collection dates of the samples. The arguments `date.from` and `date.to` are used for indicating the range of dates of samples to be retained. For instance, the range of data in `x` is:

```
> get.data(x, "date", where="samples")


[1] "2001-02-13" "2001-02-13" "2001-03-01" "2001-03-01" "2001-03-01"
[6] "2001-05-25"
```

We can retain data collected before March using:

```
> subset(x, date.to="28/02/2001")


=== obkData object ===
== @individuals==
     age sex
toto  20   m

== @samples==
    individualID sampleID       date swab sequenceID locus
1           toto        1 2001-02-13    +      No305 gene1
1.1         toto        1 2001-02-13    +      No304 gene1

== @dna==
[ 2 DNA sequences in 1 locus ]

$gene1
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No304

Base composition:
    a     c     g     t
0.305 0.261 0.127 0.307


== Empty slots ==
 @clinical, @contacts, @trees
```

Note that we have specified dates using a different format from what is used in the data. This is no issue, as date format is detected automatically by `subset`.

A third way of specifying subsets of data is using indexing of the rows of `@individuals` or `@samples`, using the arguments `row.individuals` and `row.samples`, respectively. This is particularly useful for e.g. retaining only certain type of test results, or patients within a given age class, or of a given sex. For instance, to retain only positive swabs:

```
> x@samples
```

```
    individualID sampleID       date swab sequenceID locus
1           toto        1 2001-02-13    +      No305 gene1
1.1         toto        1 2001-02-13    +      No304 gene1
2           toto        3 2001-03-01    -      No306 gene1
2.1         toto        3 2001-03-01    -    No0906S gene2
2.2         toto        3 2001-03-01    -    No0908S gene1
3           titi        2 2001-05-25    +    No0909S gene2
```

```
> get.data(x, "swab")=="+"
```

```
[1]  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

```
> subset(x, row.samples=get.data(x, "swab")=="+")
```

```
=== obkData object ===
== @individuals==
     age sex
toto  20   m
titi  67   ?

== @samples==
    individualID sampleID       date swab sequenceID locus
1           toto        1 2001-02-13    +      No305 gene1
1.1         toto        1 2001-02-13    +      No304 gene1
3           titi        2 2001-05-25    +    No0909S gene2

== @dna==
[ 3 DNA sequences in 2 loci ]

$gene1
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No304

Base composition:
    a     c     g     t
0.305 0.261 0.127 0.307

$gene2
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0909S

Base composition:
    a     c     g     t
0.306 0.265 0.126 0.303


== Empty slots ==
 @clinical, @contacts, @trees
```

Or to retain male patients only:

```
> x@individuals
```

```
         age sex
toto      20   m
John Doe  18   m
titi      67   ?
```

```
> get.data(x, "sex")=="m"
```

```
[1]  TRUE  TRUE FALSE
```

```
> subset(x, row.individuals=get.data(x, "sex")=="m")
```

```
=== obkData object ===
== @individuals==
         age sex
toto      20   m
John Doe  18   m

== @samples==
    individualID sampleID       date swab sequenceID locus
1           toto        1 2001-02-13    +      No305 gene1
1.1         toto        1 2001-02-13    +      No304 gene1
2           toto        3 2001-03-01    -      No306 gene1
2.1         toto        3 2001-03-01    -    No0906S gene2
2.2         toto        3 2001-03-01    -    No0908S gene1

== @dna==
[ 5 DNA sequences in 2 loci ]

$gene1
4 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305 No304 No306 No0908S

Base composition:
    a     c     g     t
0.306 0.260 0.126 0.307

$gene2
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No0906S

Base composition:
    a     c     g     t
0.312 0.257 0.122 0.309


== Empty slots ==
 @clinical, @contacts, @trees
```

Finally, note that several filters can be specified at the same time. For instance, we can extract data of the first individual and first locus, sampled in March or later, using:

```
> subset(x, indiv=1, locus=1, date.from="01 03 2001")
```

```
=== obkData object ===
== @individuals==
     age sex
toto  20   m

== @samples==
    individualID sampleID       date swab sequenceID locus
2           toto        3 2001-03-01    -      No306 gene1
2.2         toto        3 2001-03-01    -    No0908S gene1

== @dna==
[ 2 DNA sequences in 1 locus ]

$gene1
2 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No306 No0908S

Base composition:
    a     c     g     t
0.308 0.259 0.125 0.307


== Empty slots ==
 @clinical, @contacts, @trees
```

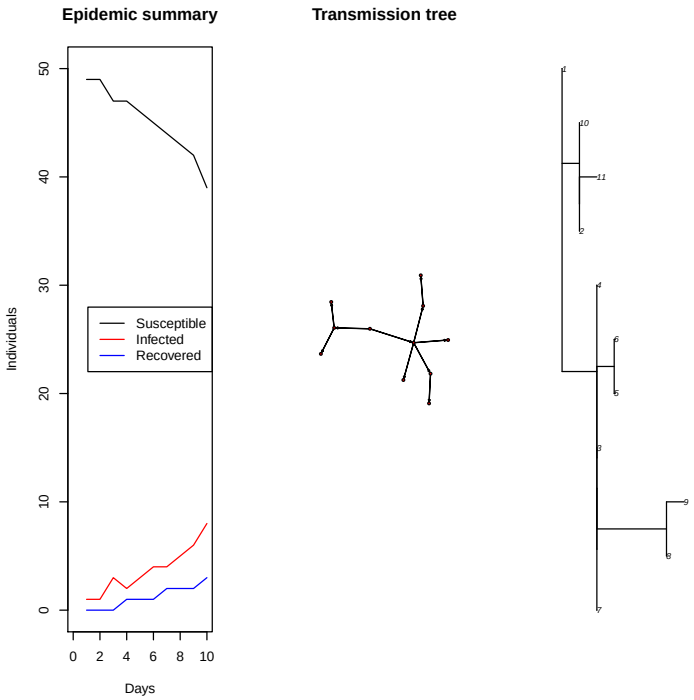
3 Simulating outbreak data

epibase provides basic functionality for the simulation of outbreak data through the `simuEpi` function. A basic SIR (susceptible-infectious-removed) model is assumed, and the result is returned as an `obkData` object. The arguments are as following:

- `N`: the size of the population, which remains constant throughout. The simulation will start with one infectious individual, `N-1` susceptibles and zero removed. Default is `N=1000`.
- `D`: duration of the simulation, in days. Default is `D=10`.
- `beta`: probability that a susceptible individual becomes infected by a given infectious individual on a given day. Default is `beta=0.001`.
- `nu`: rate of recovery, ie the probability that an infectious individual becomes removed on a given day. Default is `nu=10`.
- `L`: length of genetic sequences to be generated. Default is `L=1000`.
- `mu`: rate of mutation per site per transmission event. Default is `mu=0.001`.
- `showPlots`: logical indicating whether to plot the SIR trajectory over time, the transmission tree, and the phylogenetic tree if created. Default is `showPlots=FALSE`.
- `makePhyloTree`: logical indicating whether to create a neighbor-joining tree from the simulated sequences. Default is `makePhyloTree=FALSE`.

Let us look at an example in a very small population of size `N=50` and with the infectious rate `beta` raised accordingly to generate a few transmission events:

```
> set.seed(1);simuEpi(N=50,beta=0.01,showPlots=TRUE,makePhylo=TRUE)


=== obkData object ===
== @individuals==
   infector DateInfected
1        NA   2000-01-01
2         1   2000-01-03
3         1   2000-01-03
4         3   2000-01-05
5         3   2000-01-06
6         5   2000-01-07
7         3   2000-01-08
8         3   2000-01-09
9         8   2000-01-10
10        2   2000-01-10
11        2   2000-01-10

== @samples==
   individualID sampleID       date sequenceID
1             1        1 2000-01-01          1
2             2        2 2000-01-03          2
3             3        3 2000-01-03          3
4             4        4 2000-01-05          4
5             5        5 2000-01-06          5
6             6        6 2000-01-07          6
7             7        7 2000-01-08          7
8             8        8 2000-01-09          8
9             9        9 2000-01-10          9
10           10       10 2000-01-10         10
11           11       11 2000-01-10         11

== @dna==
[ 11 DNA sequences in 1 locus ]

[[1]]
11 DNA sequences in binary format stored in a matrix.

All sequences of same length: 1000

Labels: 1 2 3 4 5 6 ...

Base composition:
    a     c     g     t
0.244 0.275 0.232 0.249


== @trees==
1 phylogenetic trees

== Empty slots ==
 @clinical, @contacts
```



We can see that 11 individuals got infected over the default period of `D=10` days during which the outbreak was simulated. The panel on the left shows the trajectories for the number of susceptible, infectious and removed individuals over time. The panel in the middle shows the transmission tree. The panel on the right shows a Neighbor-Joining tree based on the simulated sequence data.

4 Graphics for `obkData` objects

Several plot functions are provided. These can be called by 'plot(data,'type',...)' where 'data' is an obkData object, 'type' is a string giving the type of function to plot, and optional arguments can be given. 'type' has to be one of

- 'timeline': plots the timeline of the outbreak; the timeline of every case is plotted in a single window
- 'geo' plots the cases on a map. Needs geographical information
- 'mst': plots a minimal spanning tree of the genetic data
- 'phylo': plots a phylogenetic tree of the genetic data

4.1 Plotting the timeline

Let's plot an outbreak of equine influenza provided. First import the data and make an obkData object

```
> data(HorseFlu)
> data <- new("obkData", individuals=HorseFlu$individuals,samples=HorseFlu$samples,clinical=HorseFlu$clinics)
```

then plot

```
> plot(data,'timeline')
```

or equivalently

```
> plot(data)
```

These are a lot of horses in one plot, let's restrict ourselves to a selection. We can do this by a vector specifying the indices of the individuals to plot. Lets plot the first twenty

```
> plot(data,selection=1:20)
```

Notice that the names of the individuals are now plotted. The default behaviour is to plot these when less than fifty individuals are plotted, but we can manually override this by setting 'plotNames'

```
> plot(data,selection=1:20,plotNames=FALSE)
```

The plotting of sampling times is toggled by 'plotSamples'. This defaults to TRUE, as an error will be generated when no 'date' fields can be found to plot, as would be the case for the equine dataset.

We can colour the individuals by a characteristic provided in the obkData. In this case, let's colour the horses by the yard they were in, a column called 'yardID'

```
> plot(data,selection=1:20,colorBy='yardID')
```

It might be useful to also order the individuals, use 'orderBy'

```
> plot(data,selection=1:20,colorBy='yardID',orderBy='yardID')
```

Alternatively, we could manually specify an ordering

```
> plot(data,selection=1:20,colorBy='yardID',ordering=20:1)
```

4.2 Plotting geography on a map

If geographical information is available, this function can be used to plot the cases on a map (for example downloaded from googlemaps). Geographical information can be provided as longitude/latitudes, or as strings specifying locations (which are converted to lon/lat using googlemaps).

Let's plot an outbreak of equine influenza provided. First import the data and make an obkData object

```
> data(HorseFlu)
> data <- new("obkData", individuals=HorseFlu$individuals,samples=HorseFlu$samples,clinical=HorseFlu$clinics)
```

In this dataset, we already have longitudes and latitudes. We specify the columns holding these data with 'location', and we have to tell the function that these are valid lon/lat with 'isLonLat' (which defaults to FALSE) ■= plot(data,'geo',location=c('lon','lat'),isLonLat=TRUE) We can zoom in or out using 'zoom', which is an integer (higher is zoomed in more)

```
> plot(data,'geo',location=c('lon','lat'),isLonLat=TRUE,zoom=8)
```

We can colour the individuals by a certain characteristic using 'colorBy'

```
> plot(data,'geo',location=c('lon','lat'),isLonLat=TRUE,zoom=8,colorBy='sex')
```

To center the map on an individual, use 'center'. We can use this to zoom in on a part of the map

```
> plot(data,'geo',location=c('lon','lat'),isLonLat=TRUE,zoom=12,colorBy='sex',center='9')
```

4.3 Plotting a minimal spanning tree of the genetic sequences

It can be useful to plot a minimal spanning tree of the sequences, to quickly visualize the genetic diversity and the relation between sequences. We can do this as follows

```
> data(HorseFlu)
> attach(HorseFlu)
> data <- new("obkData", individuals=individuals, samples=samples, dna=dna, clinical=clinics)
> plot(data,'mst')
```

```
[1] 1
```

this is a large tree, we can also look at the diversity within one individual

```
> plot(data,'mst',individualID=42)
```

```
[1] 1
```

4.4 Plotting a phylogenetic tree of the sequences

This function will plot a phylogenetic tree of the sequences.