EPOC

Ecosystem Productivity Ocean Climate Modelling Framework

Andrew Constable, Troy Robertson

Package: EPOC
Type: Package
Version: 0.5.2
Date: 2015-04-01
License: GPL2
Depends: methods, Rcpp
Suggest: inline

EPOC is an R-based object-oriented modeling framework built around a number of modules: Environment, Biota, Activities, Management, Output and Presentation.

R source input/data scripts dictate the creation of a model ***universe*** object containing the S4 objects which represent ecosystem ***elements***. Each element is instantiated with its associated ***attributes*** and the ***actions*** (S4 methods) that act upon themselves or each other. An EPOC ***controller*** object is able to process an EPOC universe object and an EPOC ***calendar*** of events is produced which defines the sequence of processing for the model. Models are run by the controller according to ***scenario*** constraints and across a set of described ***spatial*** domains. Output data files may be generated by element actions during the course of the simulation.

The EPOC package contains input files to construct an example universe that may be run by calling the function epoc(). See help("epoc") for more detail on this convenience function. These example attribute data and action method files can be found in the EPOC package /extdata directory. This directory can be found from R with: system.file("extdata", package="EPOC")

1. UNIVERSE

Modules and their elements are combined into an EPOC universe object which can be processed by the EPOC controller. An EPOC universe is defined by its spatial data, its report formatting and by its scenario constraints. Most importantly it is defined by its member elements. A universe data input file defines these characteristics and also points to additional data input files which define member objects. Data input files for an example universe may be found in the EPOC package /extdata directory. The spatial definition of any universe is described by a spatial input data file. It determines the areal domain for consideration during the simulation. Likewise, scenario parameters are described by a scenario data input file. These define the temporal domain being considered by the simulation. More than one scenario can be described for a simulation and these will be currently run in a serial manner in the order listed by the input file.

Each of these objects is available by using the accessor methods provided: getScenario(), getSpatial() and getReport(). Universe element members may also be accessed by action methods through the use of the getEPOCElement() method.

During a scenario run, the universe carries its own realtime state information. This information reflects the current simulation state only. This universe information can be queried by each element using the available universe S4 method getRTState(). Any universe or element state information may be written to file by an element through the use of a printState() method. If doPrint() is set to TRUE then this method will be called at the very end of any period of the simulation.

Output from the universe object can be controlled with the use of options(epocmsglevel=”normal”, epocloglevel="debug"). Other options include c(“quiet”, “normal”, “verbose”, “debug”). Output can also be controlled as arguments to to Universe constructor (see help("Universe-class"), or from within the universe data input file as a Report attribute.

2. MODULES

Each module comprises elements that are simply added to a list within its specification. Each element is an object carrying all its own methods and data. All elements are modelled in a manner similar to that described for an animal's environment by Andrewartha & Birch (1984)[i] where an element "knows" how it will interact, utilise or impact other elements, if they are there, but it does not need to know how those or other elements are going to impact upon it.

Elements are able to operate at spatial scales different to other elements with a capacity to scale one element to another within an element's specific functions. Elements can operate at scales smaller than those nominated if required. For example, some actions may require detailed small-scale modelling of surveys, foraging, fishing or monitoring. These can be based on the state of the Universe (below) with the accumulated results of the subscale actions being used to update the Universe at the appropriate time. This can be applied within polygons and/or within periods of the year.

2.1 Environment

The environment typically encompasses features of the ocean, sea-ice, atmosphere and geomorphology of benthic habitats if needed. All of these are optional as environmental variability could be incorporated into the function of biotic elements. However, the important feature of the environment module is to provide uniformity of the impacts of the environment on the biotic system. In addition, the inclusion of environmental elements means that feedback influences of biota on the physical environment can easily be incorporated in a coupled biophysical model. A simple form of environmental variability is a single element to represent variability in the advance and retreat of sea ice, such as has been used for krill. It is possible to have an element to govern vectors of ocean transport. These could then be used to vary the movement patterns of biota that are dependent on ocean vectors.

These elements may be affected by other environmental elements, biota and/or activities.

---

[i] Andrewartha, H.G., and L.C. Birch. (1984) *The ecological web: more on the distribution and abundance of animals*. University of Chicago Press, Chicago.

2.2 Biota

Biotic elements can have varying characteristics (Constable, 2005b)[ii] including:

a. an age-structured population,
b. a life-stage and/or sex of a population (which may or may not be age-structured),
c. an individual animal (if you need that sort of detail), and
d. a guild of one or more taxa modelled as a single biomass or numerical unit.

Clearly, life stages need to interact with other life stages. In such cases, different life stages could be modelled as separate elements (see below).

This structure potentially gives the model similarities to an individual-based (agent-based) model although it need not be structured in that way if it is not required.

These forms of biota provide opportunities for modelling at a general population level or subdividing the population into different groups because of their different breeding, foraging and/or growth characteristics even though they may reside or forage in similar locations to one another. For example, there may be benefit in using separate sex models for some fish stocks. Similarly, separating colonies of land-based predators might be useful to enable them to forage widely and in the same locations as other colonies but returning to their respective colonies to breed. Exchange between colonies can also be readily modelled.

These elements may be affected by other biotic elements, the environment and/or activities.

2.3 Activity

This module comprises all human activities to be simulated. These could include fishing, scientific research, surveys, ecosystem monitoring programs and so on. These activities would be expected to affect the state of the Universe in some way, although some may be considered to be negligible. The primary activities being considered at present are:

a. surveys of fish and krill stocks,

[ii] Constable, A.J. (2005b) A possible framework in which to consider plausible models of the Antarctic marine ecosystem for evaluating krill management procedures. CCAMLR Science 12:99-117.

b. fisheries, and
c. monitoring of land-based predators.

Elements can be constructed for these activities in a similar way to biotic elements. These activities may affect each other and/or, importantly, management decisions, as well as be affected by the environment and biota.

2.4 Management

The management module comprises actions and activities that occur away from the ecosystem. Typically, the management module would only gain information from Human activities. These data can be used in assessments and decision-making. In the scenario developed below, the elements of management are:

a. assessment of long-term annual yield arising from surveys, and
b. setting of catch limits by area and time.

The State of Management provides the information necessary to constrain activities. The most efficient form of prescribing elements of Management will be to have at least one element in Management referred to by each Human activity. Thus, for the activities described above the elements of management would be:

a. governing rules for surveys of fish and krill,
b. catch limits by area and time for each fishery, and
c. governing rules for monitoring land-based predators.

2.5 Output

The outputs from the simulation do not affect any of the above modules. Most elements will output there own data at the prescribed times during a simulation. However, some reporting of the state of the universe may need to be an amalgam of data. This module would be used for those purposes.

2.6 Presentation

As for Output, this module has been provided as a means of producing tables, graphs and summarised data reporting.

3. ELEMENTS

EPOC elements are the basic building blocks of an EPOC model. They are described by a set of attributes and a set of actions. Just as input data files are used to define and structure a model universe, input data files are also used to facilitate instantiation of an element. An element may be represented by an EPOC attribute data input file and one or more files containing EPOC action S4 methods. When instantiated, an element will load its input attributes from the file path passed. All EPOC actions referred to by the element will also be sourced from the file paths defined in the input file.

An EPOC element (like the universe object) stores it current state. Additionally it stores transition state, that is, the modified state data values which have occurred 'During' the period. During the updateState() method call which occurs between the 'During' and 'After' timestep timing, the elements state is modified with respect to the transition state and the transition state re-zeroed if necessary. Both state and transition state data is made available to action methods via accessors: getState(), setState(), getTransition() and setTransition().

An elements timestep information will be combined with that from all other member elements of the universe and used by the EPOC controller to construct a yearly calendar of actions.

4. ACTIONS

EPOC elements can have four different types of action methods associated with them:

Default – Four default methods are enabled which will be called if available at predetermined points in the simulation. These are: initialiseReplicate, initialiseTransition, updateState and printState. The first two are called before commencement of the simulation. They can be used to perform initialisation tasks on the element object with respect to the scenario parameters and transition state. An updateState method will be executed between the "During" and "After" timestep timings if the doUpdate flag has been set for the element (using: doUpdate(element, TRUE)). Any printState method will be executed in the same way but as the last step before the period

ends and only if the doPrint flag is set to TRUE (using: doPrint(element, TRUE)). Writing to file connections at the end of every period can be costly. An alternative is to shedule the printState method as a timestep action method. To ensure the final state is output, a flag doPrintFinal (using: doPrintFinal(element, TRUE)) is available which will force the execution of the printState method as the last step before ending the scenario.

Transform (Setup) – Allows period transformation or setup operations to be carried out during the calendar creation process.

Support – Standalone methods which are callable by other action methods. This allows for the organisation of code into smaller, more reusable and maintainable units.

Timestep – These are element action methods which are to be incorporated into the simulations calendar of events. They are associated with a temporal period. Each timestep action can have a number of attributes defined:

- actionMethod - the name of the action method
- actionFile - the path to the file containing the source code for the method.
- tsType - the timestep type ("FirstPeriod", "LastPeriod", "AllPeriods")
- tsTiming - the timestep timing ("Before", "During", "After")
- transAction - specifies a setup action with associated name, path and dataset.
- relatedElements – a matrix of name/module pairs of related elements which will be referenced by the action method.
- dset – a dataset or attribute value which can be made available to the method.

Once an element is instantiated, the timestep list is available via the getTimestep method.

5. EPOCObject

All objects in EPOC, whether they be the universe or an element object are represented as S4 class objects that contain (inherit) the EPOCObject class. This is the base VIRTUAL class on which most EPOC methods are defined. All EPOCObjects are identified by a Signature object. The accessor method getSignature() and formatted methods getSignatureLine() and getSignatureMulti() methods are available. All EPOCObjects are able to source a list of input data upon instantiation via the dataPath argument to their initialze method. The object will store any EPOC Attributes read from the data input file.

EPOC Attributes will be stored preferentially in slots of the same name and secondarily in a list stored in the 'epocAttributes' slot. Any object of this class then make available the getAttribute() accessor to action method code. Base method getSlot() is also available for any other object slots but it is preferable to use specific accessor methods for the slots in question. All slot data is intended to be static data once loaded by the object. Transient data such as state, transient, functions, flag lists are stored within an R environment inherited by all EPOCObjects at .xData. This allows modification of an objects data by Action methods during a simulation without the need for explicit object returns or the usual R on-change memory copy costs. See help("EPOCObject-class")

6. DATA FILES

Data input files are used to describe a the instantiation of a single EPOCObject and are structured as an R list object returnable as an expression. Each list element matches either a named slot or, if not, then it will be used to represent an attribute of the object being created. Generally it is best to declare an R list object and then assign named list elements singly. This allows reference to earlier listed attributes by later ones in order to save duplication of input data (eg polygonsN). After the list object is declared it is generally followed by a signature list object which will be instantiated as a Signature object in its own right and attached as a data member of the EPOCObject being created.

```
MyObject <- list()
MyObject$signature <- list(
```

```
        ID          = 15001,
        ClassName   = "Biota",
        Name.full   = "My Big Thing",
        Name.short  = "MBT",
        Morph       = "Things",
        Revision    = "01",
        Authors     = "T. Robertson",
        Last.edit   = "20 Oct 2010"
)
```

6.1 Universe

A universes data input file requires an inputPaths list which is broken into sublists by EPOC module name and then by EPOC element name. Each EPOC element is represented by a list containing className, classFile and classData. The className specifies the name of the class to instantiate with the data from the input data file. This may either be one provided by the EPOC R package or be pointed to by the following list element, classFile. The classData list element provides a file path to the data input file itself.

Included in the inputPaths list is a 'Config' list element. This is itself a list of configuration objects 'Polygons' and 'Scenarios' which hold a file path to their respective data input files.

```
Universe <- list()
Universe$inputPaths <- list(
    Config = list(
        Polygons    = file.path("data", "Spatial.data.R"),
        Scenarios     = file.path("data", "Scenario.data.R")
    ),
    Environment = list(
        KrillEnv = list(className = "KrillEnvironment",
            classFile = file.path("code", "E.O.Env.KPFM.01.R"),
            classData = file.path("data", "E.O.Env.KPFM.data.01.R"))
    ),
    Biota = list(
        Krill = list(className = "Krill",
            classFile = file.path("code", "B.MI.Es.KPFM.01.R"),
            classData = file.path("data", "B.MI.Es.KPFM.data.01.R")),
        PenguinsA3 = list(className = "Predator",
            classFile = file.path("code", "B.Pr.KPFM.01.R"),
            classData = file.path("data", "B.Pr.KPFM.01.data.PenguinRecAge3.01.R"))
    )
)
```

Additionally, the universe data input file can contain, as per all data input files, zero or more attribute values that will be stored in a similarly named slot if one exists, else in the universes epocAttributes slot. This includes the report formatting values. See help("Universe-class")

6.2 Spatial

The spatial data input file requires inclusion of six list elements: polygonsNames, polygonAreas, coords, coordAreas, coordProportions and overlap. Except for the last, each of these is a list of those values for each polygon available in the model universe. The last, 'overlap', is a 2d matrix of all polygons by all polygons showing a 1 where those two polygons coincide. See help("Spatial-class")

6.3 Scenario

A single scenario data input file may describe one or more scenarios to be performed in a consecutive manner. The scenario data input file is layed out slightly differently in that the input list contains each scenario as a list element. Each scenario must contain at least a signature list, yearStart, yearEnd and scenarioDir attributes. Output files will be written by default to this scenarioDir path. It may of course also include any additionally required attributes that may be queried by action methods. See help("Scenario-class")

```
Scenarios <- list()
Scenarios$FirstScenario <- list()
Scenarios$FirstScenario$signature <- list(
        ClassName       = "Scenario",
        ID              = 12002,
        Name.full       = "KPFM Scenario parameters 0801",
        Name.short      = "KPFM.Scenario01",
        Morph           = "",
        Revision        = "1",
        Authors         = "Troy Robertson",
        Last.edit        = "19/10/2010 11:21:44 AM"
)

Scenarios$FirstScenario$yearStart           <- 1950
Scenarios$FirstScenario$yearEnd             <- 1960
Scenarios$FirstScenario$yearsN              <- 11
Scenarios$FirstScenario$firstFishingYear    <- 1952
Scenarios$FirstScenario$lastFishingYear     <- 1956
Scenarios$FirstScenario$scenarioDir         <- file.path(getwd(), "runtime")
Scenarios$FirstScenario$replicateCnt        <- 1
```

6.4 Element

There are a number of list elements required in any EPOC element data input file, as well as the usual signature list. A polygons vector is required which contains at least one corresponding universe polygon index. Additionally the list must contain a 'birthdate' element which is itself a list containing two values, day and month. This birthday represents the time of the year which is the start time of an annual cycle for the element. The polygons vector and birthday can be accessed via the getPolygons() and getBirthday() methods.

The element input data file must also include a list element 'Timesteps' which contains a list of timesteps. See Section 3 for a description of the timestep attributes. Any element data input file from the /extdata package directory will give an example of the format required. See help("Element-class")

7. ACTION/METHOD FILES

7.1 Default methods

Four default methods exist for any EPOC element as described in section 4. To override these methods (or create user defined ones for a new module element class) the following signature is required.

```
setMethod("initialiseReplicate", "MyElement",
        function(.Object="ElementClassName", universe="Universe") {
                # initialise MyElement
                ...
                return(.Object)
        }
)
```

The same applies for setTransition. The updateState and printState methods do not need to return(.Object) as both state and transition state data is stored in an R environment (.xData) allowing pass by reference.

7.2 Transform (setup) methods

Transform methods which are used to setup or modify timestep data during calendar creation dependent on allocated period information. They currently have a tedious set of parameters. The isGeneric() call is a good idea as it deals with situations where a generic method has already been declared for that method name.

```
if (!isGeneric("mySetup"))
setGeneric("mySetup ", function(.Object, period="Period", ptSA="list", modulenum="numeric",
elementnum="numeric", firstperiod="numeric", dset="list") standardGeneric("mySetup "))
setMethod("consumeSetup", "MyClass",
    function(
        .Object,            # element environment
        period,             # period
        ptSA,               # untransformed action for the period derived from timestep of element
                            # Note: PtSA is a list retained for concatenating to a list.
                            #       Therefore, the action is the first element in the list
        modulenum,          # reference module number for the element
        elementnum,         # relative number of the element in the universe
        firstPeriod,        # logical indicating if this is the first period in the timestep
        dset                # dataset to assist with transformation
                )
    {
        # perform transformation
        return(ptSA)
    }
)
```

7.3 Timestep methods

Timestep methods are the mainstay of a EPOC model representing actions performed by an EPOC element which modify that elements transition state data and/or that of other elements. It is important to remember to use unique method/action names for each element class. Duplicates will override the definition of previous methods of the same name. Timestep methods must follow the following signature:

```
if (!isGeneric("myAction"))
setGeneric("myAction ", function(.Object, universe) standardGeneric("myAction "))
setMethod("myAction ", signature(.Object="MyClass", universe="Universe"),
    function(.Object, universe) {
        # Do stuff
    }
)
```

If printState() or updateState() methods are to be used as timestep action methods then, as default methods, they do not require isGeneric() and setGeneric statements.

For any method, additional EPOC attributes or universe realtime state information can be accessed using appropriate methods on the parameters passed.

e.g.

```
# Get a handle on some necessary universe and element state data
# This current action matrix row
action <- getRTState(universe, "currentAction")
                    # ActionMat row
                    # Col  1  = module
                    # Col  2  = element
                    # Col  3  = period
                    # Col  4  = reference day in year
                    # Col  5  = action reference number in period (NA if no actions)
                    # Col  6  = number for "before =1", "during = 2", "after = 3" (NA if no actions)

# Calendar periodInfo
periodInfo <- getRTState(universe, "currentPeriodInfo")
                    # periodInfo  # information about the active period for use in subroutines
                    # Number      = eTSD
                    # Day         = PropYear[eTSD,1]
                    # KnifeEdge   = if(PropYear[eTSD,2]==0) FALSE else TRUE
                    # YearPropn   = PropYear[eTSD,3]
                    # PeriodStart = PreviousDay/365 # proportion of year passed since 0 Jan
                    #                     # to beginning of time period
                    # PeriodEnd   = PreviousDay/365+PropYear[eTSD,3]

# Current scenario number
scenarioNum <- getRTState(universe, “currentScenario”)

# Current year
currYear <- getRTState(universe, “currentYear”)

# Current relative year
yearNum <- getRTState(universe, “relativeYear”)

# Current period number
periodNum <- getRTState(universe, “currentPeriod”)

# Element state list
elemState <- getState(.Object)

# Element transition state list
elemTrans <- getTransition(.Object)

# Element timestep data for this action
elemTS <- getTimestep(.Object, action[3])$actions[[action[5]]]

# This action timestep dataset
dset <- elemTS$dset
```

7.4 Support methods

These are methods callable by other methods whether they be default, transform or timestep methods. They just provide a way to break up code into more reusable and manageable chunks. As such there is no interface proscribed and it is left to the user to ensure that calls match parameter signatures for any support methods.

8. C++ ACTION METHODS

Since version 0.5.0 of EPOC, a shared library is provided which makes available most EPOC API methods. This allows a user to write Element action methods utilising C++ code, and to perform a runtime compile and to then access that method using a combination of the Rcpp and inline packages, and a simple wrapper method setEPOCCPPMethod(). See help("setEPOCCPPMethod") for more detail. This method is most appropriate for EPOC model development and prototyping situations.

If a number of C++ functions are to be included in a model, it is advisable to pre-compile all functions into a shared library once the development phase is concluded. This can be done simply by ensuring rTools is installed and by using RCMD SHLIB at the command line. Compiled library functions can then be included in a model in the same way as R source file actions. A timestep action needs only to have its actionMethod named to the function, and the filePath to point to the shared library. Any filePath to actionMethod that is not to a file named *.R will be assumed to be to a shared library. An alternative is to wrap the C++ .Call within an R S4 method which makes the C++ transparent to EPOC. The function setEPOCLibMethod() is provided for this purpose. See help("setEPOCLibMethod").

The EPOC_API vignette provides details on the EPOC methods available from within EPOC action methods in both R and C++, see vignette("EPOC_API"). An example of the use of inlined C++ action methods can be found in the EPOC packages example

model file.path(system.file("extdata", package="EPOC"), "code",
"B.Pr.KPFM.Consume.cpp.01")

Alternatively, to examine the EPOC library in more detail, as header file is available in the packages 'includes' directory, and the complete source is available in the EPOC R source package.

9. FILE MANAGEMENT

In order to facilitate faster access to output data files, the EPOCObject class provides a number of file methods. See help("EPOCObject") for more links to more detail on these methods. A list of file connections is maintained for each EPOCObject. When opened they are stored in a list for further reads or writes. When the simulation finalises, all file connections will be closed. A closed file connection is still retained in the list and further reads or writes will first force an opening of the file. This same mechanism is used for the universe log file which has a copy passed to all EPOCObjects. EPOCObjects will not only write all epocMessage to stdout, but will also write text to the logfile if loglevel settings permit.

File connections do not utilise the standard R connection mechanism. The EPOC shared library has an underlying FileConnection class which uses std::fstream C++ methods to open, read, write and close file connections. The EPOC C++ API provides hooks for the EPOC R methods to be able to call these underlying functions. The list of file connections managed by each EPOCObject in R is a list of externalptr(s). These externalptr(s) can be passed across the R/C++ interface. This allows other EPOC C++ API functions to utilise the same file connections via these externalptr(s). EPOC action methods written in C++ can then utilise these same C++ file connection methods. See vignette("EPOC_API") for detail on these functions.

10. EXAMPLE MODEL – FOOSA

An operating food web model for evaluating spatially-structured harvest strategies for krill.

Examples

```
# Set the root working directory for input/output files.
setwd(...)

# The EPOC package contains a set of example input data files and action methods.
# These can be run simply by instantiating the convenience function epoc().  This
# function will load input files create a model universe and run the simulation just by:
e <- epoc()


# Alternatively a user created universe will be run by:
e <- epoc(dataPath=file.path(getwd(), "data", "Universe.data.R"))


# OR to skip the convenience.
# Create a model universe using passed input data file.
universe <- new("Universe", dataPath=file.path(getwd(), "data", "Universe.data.R"))

# Start controller - this creates/sets up both the universe and calendar
controller <- new("Controller", universe=universe, outputcalendar=TRUE, tofile=TRUE)

# Start simulation
runSimulation(controller, timer=TRUE)
```