Package 'AssetPricing'

October 12, 2022

Version 1.0-3

Date 2021-10-08

Title Optimal Pricing of Assets with Fixed Expiry Date

Author Rolf Turner <r.turner@auckland.ac.nz>

Maintainer Rolf Turner <r.turner@auckland.ac.nz>

Depends R (>= 0.99)

Imports polynom, deSolve

Description Calculates the optimal price of assets (such as airline flight seats, hotel room bookings) whose value becomes zero after a fixed ``expiry date''. Assumes potential customers arrive (possibly in groups) according to a known inhomogeneous Poisson process. Also assumes a known time-varying elasticity of demand (price sensitivity) function. Uses elementary techniques based on ordinary differential equations. Uses the package deSolve to effect the solution of these differential equations.

License GPL (>= 2)

URL http://www.stat.auckland.ac.nz/~rolf/

NeedsCompilation no

Repository CRAN

Date/Publication 2021-10-07 20:40:11 UTC

R topics documented:

`buildS` *Build a piecewise linear price sensitivity function*

Description

Builds a price sensitivity function which is piecewise linear in price, in an automated manner, with built-in checks for possible infelicities.

Usage

```
buildS(alpha, beta, kn, tmax)
```

Arguments

`alpha`	A list of functions of `t` giving the constant terms of the linear functions comprising the price sensitivity function.
`beta`	A list of functions of `t` giving the slopes of the linear functions comprising the price sensitivity function.
`kn`	The knots (with respect to price) of the piecewise linear price sensitivity function. The zero knot (which is always the first knot) is *not* included in `kn`.
`tmax`	The maximum time value to which the price sensitivity function is to be applied. Needed for internal consistency checks.

Details

The price sensitivity function is assumed to be of the form

$$S(x,t) = \alpha_k(t) + \beta_k(t)x$$

for $x_{k-1} \leq x \leq x_k$ where $x_1, x_2, \ldots, x_K$ are the (non-zero) knots of the function. It is assumed that $x_0 = 0$. The variable `x` represents price and the variable `t` represents residual time.

The function is defined over the rectangle $[0, x_K] \times [0, t_{\max}]$.

Checks are done to make sure that

- $S(x,t)$ **is continuous**
- $S(0,t) = 1$ **for all** t
- $S(x,t)$ **is non-increasing in** x **for all t**
- $S(x,t) \geq 0$ **for all** x **and** t

Value

A function of two variables `x` and `t`, which is a price sensitivity function. The argument `x` represents price and the argument `t` represents (residual) time. The value of the function is interpreted as the probability that a customer "arriving" at time `t` will purchase an item offered at price `x`.

Author(s)

Rolf Turner <r.turner@auckland.ac.nz> http://www.stat.auckland.ac.nz/~rolf

References

P. K. Banerjee, and T. R. Turner (2012). A flexible model for the pricing of perishable assets. *Omega* **40**:5, 533–540. DOI https://doi.org/10.1016/j.omega.2011.10.001

Rolf Turner, Pradeep Banerjee and Rayomand Shahlori (2014). Optimal Asset Pricing. *Journal of Statistical Software* **58**:11, 1–25. DOI https://doi.org/10.18637/jss.v058.i11

See Also

xsolve()

Examples

```
   lambda <- function(t) {
      tn <- 1:4
      A <- matrix(c(0,12,12,12,
              0,-16,16,64,
              20,30,30,0),nrow=4)
      B <- matrix(c(12,0,0,0,
               0,16,0,-16,
               0,-10,-10,0),nrow=4)
      s <- cut(t,breaks=c(0,tn),include.lowest=TRUE,labels=tn)
      s <- as.numeric(levels(s)[s])
      M <- matrix(A[s,] + B[s,]*t,ncol=ncol(A))
      M[!is.finite(M)] <- 0
      M
   }

   alpha <- vector("list",4)
   beta  <- vector("list",4)
   alpha[[1]] <- with(list(lambda=lambda),
function(t) {
A <- c(1,1,1)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%A/dnm
   })
   beta[[1]] <- with(list(lambda=lambda),
function(t) {
B <- c(0,0,0)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%B/dnm
   })
   alpha[[2]] <- with(list(lambda=lambda),
function(t) {
```

```
A <- c(1.495,1,1)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%A/dnm
   })
   beta[[2]] <- with(list(lambda=lambda),
function(t) {
B <- c(-0.2475,0,0)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%B/dnm
   })
   alpha[[3]] <- with(list(lambda=lambda),
function(t) {
A <- c(0.01,2.485,1)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%A/dnm
   })
   beta[[3]] <- with(list(lambda=lambda),
function(t) {
B <- c(0,-0.2475,0)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%B/dnm
   })
   alpha[[4]] <- with(list(lambda=lambda),
function(t) {
A <- c(0.01,0.01,3.475)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%A/dnm
   })
   beta[[4]] <- with(list(lambda=lambda),
function(t) {
B <- c(0,0,-0.2475)
        lll <- lambda(t)
        dnm <- apply(lll,1,sum)
        dnm[dnm==0] <- 1
        lll%*%B/dnm
   })
   kn <- c(2,6,10,14)
   S  <- buildS(alpha,beta,kn,4)
   x  <- seq(0,14,length=41)
   t  <- seq(0,4,length=41)
   z  <- S(x,t)
## Not run:
   persp(x,t,z,theta=150,phi=40,d=4,xlab="price",ylab="time",
```

```
          zlab="probability",ticktype="detailed")

## End(Not run)
```

`plot.AssetPricing` *Plot a list of asset pricing functions.*

Description

Plot a list of functions — in particular optimal price functions or expected value functions or derivatives of the expected value functions. Such a list is assumed to occur as a component of an object produced by `xsolve()` of `vsolve()`. The functions in the list are functions of residual time. The indices of the list correspond to the number of items available for sale and possibly (for optimal price functions) the size of the arriving group of customers.

Usage

```
## S3 method for class 'AssetPricing'
plot(x,witch=c("price","expVal","vdot"),
             xlim=NULL,ylim=NULL,lty=NULL,cols=NULL,xlab=NULL,
             ylab=NULL,main=NULL,main.panel= NULL,groups=NULL,
             add=FALSE,gloss=FALSE,glind=NULL,extend=0.3,col.gloss=1,
             cex.gloss=0.8,mfrow=NULL,...)
```

Arguments

`x`	An object of class `AssetPricing`, i.e. an object produced by `vsolve()`, or `xsolve()`.
`witch`	A text string indicating which of the three possible components of `x` should be plotted. May be abbreviated, e.g. to `p`, `e` or `v`.
`xlim`	The `x` limits of the plot. Defaults to the `tlim` attribute of the object `x[[witch]]`. If this attribute does not exist and `xlim` is not supplied then an error is given.
`ylim`	The `y` limits of the plot. Defaults to the `ylim` attribute of the object `x[[witch]]`. If this attribute does not exist and `ylim` is not supplied then an error is given.
`lty`	A vector of line types. It will be replicated to have a length equal to the number of rows of `groups` (see below). Defaults to having all entries of the vector equal to `1`, i.e. solid lines.
`cols`	A vector of colours for the plotted lines. It will be replicated to have a length equal to the number of rows of `groups` (see below). Defaults to having all entries of the vector equal to `1`, i.e. black.
`xlab`	A text string giving a label for the `x` axis (or axes). Defaults to the null string. Ignored if `add` is `TRUE`.
`ylab`	A text string giving a label for the `y` axis (or axes). Defaults to the null string. Ignored if `add` is `TRUE`.

main	A text string giving an overall title for the plot or for each page of plots if there is more than one. Defaults to the null string and is ignored if add is TRUE.
main.panel	A text string which is replicated "np" times (where "np" is the total number of panels) or a vector of text strings of length equal to "np". Note that "np" will be equal to the number of unique entries of groups$group. (See below.) The i^{th} entry of the vector is used as the title of the i^{th} panel of the plots that are created. If main.panel is left NULL the i^{th} entry of the vector is set equal to paste("group",i). This argument is ignored if there is only a single panel.
groups	A data frame with one, two or three columns, named group, q and j. The total number of rows should be less than or equal to the total number of entries of the function list x[[witch]]. Only those function traces corresponding to a row of groups are plotted. The traces corresponding to an individual value in the group column are plotted in the same panel of a multi-panel array of plots. See **Details**.
add	Logical scalar; should the plot be added to an existing plot?
gloss	Either a logical scalar (should a "marginal gloss" be added to the plot? — if TRUE then the gloss is constructed internally; see **Details**) or a vector of character strings of which the marginal gloss is to consist.
glind	A logical vector indicating which entries of gloss should actually be used (plotted). I.e. marginal gloss is added for the graphs of functions whose corresponding values in the entries of glind are TRUE. Ignored if gloss is FALSE. If gloss is TRUE or is explicitly provided, then if glind is not specified it defaults to a vector, of the same length as gloss all of whose entries are TRUE.
extend	A scalar, between 0 and 1, indicating how much the x-axis should be extended (to the right) in order to accommodate the marginal gloss.
col.gloss	Scalar specifying the colour in which the marginal gloss is to be added, e.g. "red" (or equivalently 2). The default, i.e. 1, is black.
cex.gloss	Character expansion (cex) specifier for the marginal gloss.
mfrow	The dimensions of the array(s) of panels in which the functions are plotted. If this argument is left as NULL then the software makes a "sensible" choice for its value. If this argument is set equal to NA then the current value of mfrow for the plotting device is left "as is". This permits the setting up of an array of panels vi a call to par(mfrow=...) *a priori* without the resulting setting being overridden by the internal code of this plotting method. One might wish to do this e.g. for the purpose of *adding* plotted material to each panel.
...	Extra arguments to be passed to plot (effectively to plot.function() or to plot.stepfun()).

Details

If the argument groups is specified then:

- it must *always* have a column q. The values in this column should be integers between 1 and qmax (see below).
- if jmax (see below) is greater than 1 it must also have a column j. The entries of this column should be integers between 1 and jmax.

- if jmax is equal to 1 then column j need not be present. In this case, it is internally set equal to a column of 1-s.
- if the group column is present its entries should be (consecutive) positive integers running from 1 to the total number of groups.
- if the group column is not present then this column is internally set equal to a column of 1-s i.e. there is a single group of traces.

The value of qmax is the maximum number of items that are available for sale in the time period under consideration. It may be obtained as attr(x,"qmax").

The value of jmax is, when "double indexing" applies, the maximum size of an arriving group of customers, and is otherwise equal to 1. It may be obtained as attr(x,"jmax"). Note that "double indexing" can *only* apply when x[[witch]] is a list of *price* functions, i.e. when witch is equal to price. Hence "double indexing" does not apply when witch is equal to expVal or to vdot. In these cases jmax is equal to 1.

If groups is not specified then it defaults to a data frame with number of rows equal to the length of x[[witch]], The group column has entries all equal to 1, i.e. there is a single group of traces. The q and j columns contain all possible (valid) combinations of stock size and customer group size.

If gloss is FALSE then no marginal gloss is plotted. If gloss is TRUE then the marginal gloss is created from the values of the q and j entries in the columns of groups using paste().

Note that if add is TRUE then the gloss may not actually appear in the plot, since it is placed at the right hand edge of the plot and may consequently be outside of the plotting region. Thus if you wish to use a gloss when adding to an existing plot you will probably need to take steps to ensure that there is room in the right hand margin for the plot to appear, or possibly set par(xpd=NA).

If "double indexing" applies then x[[i]] corresponds to a stock size of q and a customer group size of j where i = (j-1)*(qmax - j/2) + q.

To get traces plotted in individual panels (one trace per panel) set the group column of groups to be 1:n where n is the total number of traces being plotted.

This function (i.e. plot.AssetPricing() calls upon an "internal" function plot.flap() to do the hard yakka. (Note that flap stands for dQuotefunction list for asset pricing.)

The function plot.flap() makes use of a modified version of plot.stepfun(), rather than the one which appears in package:stats. The modification causes plot.stepfun() to treat the xlim argument in a manner similar to the way in which it is treated by plot.function. Note that plot.stepfun() is *not* exported from this package. On the advice of Kurt Hornik (31/03/2018) I created a new generic plot() function in this package (i.e. AssetPricing) with default method equal to graphics::plot(), so as to properly accommodate the existence of this modified plot.stepfun() method.

Value

None. This function exists only for its side effect, i.e. the production of a plot or plots.

Author(s)

Rolf Turner <r.turner@auckland.ac.nz> http://www.stat.auckland.ac.nz/~rolf

References

P. K. Banerjee, and T. R. Turner (2012). A flexible model for the pricing of perishable assets. *Omega* **40**:5, 533–540. DOI https://doi.org/10.1016/j.omega.2011.10.001

Rolf Turner, Pradeep Banerjee and Rayomand Shahlori (2014). Optimal Asset Pricing. *Journal of Statistical Software* **58**:11, 1–25. DOI https://doi.org/10.18637/jss.v058.i11

See Also

`xsolve()` `vsolve()`,

Examples

```
## Not run:
S <- expression(exp(-kappa*x/(1+gamma*exp(-beta*t))))
attr(S,"parvec") <- c(kappa=10/1.5,gamma=9,beta=1)
LAMBDA <- function(tt){
    if(tt<0 | tt> 1) 0 else 36*(1-tt)
}
OUT <- xsolve(S=S,lambda=LAMBDA,gprob=(5:1)/15,tmax=1,qmax=30,
                  alpha=0.5,type="dip",verbInt=2)
GLND <- rep(FALSE,30)
GLND[c(1:5,10,15,20,30)] <- TRUE
plot(OUT,witch="e",xlab="residual time",ylab="expected revenue",
     gloss=TRUE,glind=GLND)
GRPS <- data.frame(group=rep(1:6,each=5),q=1:30)
GLND <- c(TRUE,FALSE,TRUE,FALSE,TRUE,rep(c(rep(FALSE,4),TRUE),5))
plot(OUT,witch="e",groups=GRPS,xlab="residual time",ylab="expected revenue",
     gloss=TRUE,glind=GLND)
GRPS <- data.frame(group=rep(1:5,each=6),j=rep(1:5,each=6))
GRPS$q <- with(GRPS,pmax(j,rep(c(1,6,11,16,21,26),5)))
GLND <- rep(c(TRUE,TRUE,rep(FALSE,3),TRUE),5)
plot(OUT,witch="p",groups=GRPS,mfrow=c(3,2),gloss=TRUE,glind=GLND,xlab="price")
# Pretty messy looking:
GRPS$group <- 1
GLND <- unlist(lapply(1:5,function(k){(1:6)==k}))
plot(OUT,witch="p",groups=GRPS,gloss=TRUE,glind=GLND,cols=GRPS$j,xlab="price")

## End(Not run)
```

`vsolve` *Solve for expected value of assets.*

Description

Solves a system of coupled differential equations for the expected value of a number `q` of ("perishable") assets, with `q` running from `1` to `qmax`, *given* a pricing policy. Treats the system in a vectorized form and uses the method of Runge-Kutta.

Usage

```
vsolve(S, lambda, gprob, tmax=NULL, x, nout=300,
       alpha=NULL, salval=0, method="lsoda",verbInt=0)
```

Arguments

S	An expression, or list of expressions, or a function or list of functions, specifying the price sensitivity functions S_j(x,t). See **Details**.
lambda	A function (of residual time t — see tmax) or a positive constant specifying the intensity of the (generally inhomogeneous) Poisson process of arrival times of groups of potential customers.
gprob	A function (to calculate probabilities) or a numeric vector of probabilities determining the distribution of the size of an arriving group of customers. Must be compatible with certain characteristics of s (see below). See **details**.
tmax	The maximum residual time; think of this as being the initial time at which the assets go on sale (with time *decreasing* to zero, at which point the value of each asset drops to the "salvage value" (salval), usually 0). The system of differential equations is solved over the time interval [0,tmax]. See **Details**.
x	An object of class flap (see xsolve()) specifying the (given) pricing policy. It has the form of a list of functions x_i(t), with i running from 1 to qmax if x is "singly indexed", i.e. *not* of class di.flap or x_ij(t), with i running from 1 to qmax and j running from 1 to the maximum group size if x is "doubly indexed", i.e. *is* of class di.flap. Note that x has (must have) an *attribute* qmax specifying the maximum number of assets available for sale, i.e. the number of assets available at the starting (residual) time tmax.
nout	The number of points at which values of the solution are to be provided. These are taken to be equispaced on [0,tmax].
alpha	A numeric scalar between 0 and 1 specifying the probability that an arriving group of size j > q (where q is the number of assets remaining for sale) will *consider* purchasing (all of) these remaining assets. It is irrelevant (and defaults to 1 as a "place holder") if customers always arrive singly.
salval	A (non-negative) numeric scalar specifying the "salvage value" of an asset — i.e. the quantity to which the value of an asset drops at residual time t=0. Usually salval is equal to 0.
method	Character string specified the solution method to be used by the differential equation solver ode. There is a fairly large number of possible methods. See ode() for details.
verbInt	A scalar value which controls "verbosity". If verbInt > 0 then a "progress report" is printed every verbInt *seconds* (roughly). See xsolve() for a bit more detail.

Details

The components of the argument S may be provided either as expressions or functions. If the former, these expressions should be amenble to differentiation with respect to x and t via the function deriv3(). This is essentially a matter of convenience; the derivatives are not actually used by

vsolve. The expressions are turned into functions by deriv3() in the same manner as is used by xsolve(). See the help for xsolve() for further information about the required nature of S.

The argument tmax (if specified) must be less than or equal to the tmax attribute of argument S if S is a piecewise linear price sensitivity function, and must also be less than or equal to the tlim attribute of argument x.

If tmax is not specified it will be set equal to the tmax attribute of argument S if S is a piecewise linear price sensitivity function, in which case this attribute must be less than or equal to the tlim attribute of argument x. (If this is not so then S and x are incompatible.) Otherwise tmax will be set equal to the tlim attribute of argument x.

The argument gprob determines the range of possible values of the size of an arriving group of customers. The maximum value of this group size is in effect that value of j for which the corresponding probability value is numerically distinguishable from zero. If the argument x is a "doubly indexed" list of functions (was created with type="dip") then the maximum value of group size as determined by gprob must be compatible with the indexing scheme of x. That is to say, it must be less than or equal to the jmax attribute of x, otherwise an error is given. Note that if single indexing is in effect (i.e. x was created with type="sip") then this attribute is equal to 1, but for single indexing x does not depend on group size and so no restriction is imposed.

Value

A list with components

x	The argument x which was passed to vsolve, possibly with its tlim attribute modified. It is an object of class flap.
v	An object of class flap whose entries are (spline) functions v_q(t) specifying the expected value of q assets at time t as determined by numerically solving the coupled system of differential equations.
vdot	An object of class flap whose entries are the derivatives (with respect to t) of the functions v_q(t) described above. The values of these derivatives are determined as the left hand side of the differential equations being solved.

Note

A substantial change was made to this package as of the change of version number from 0.0-11 to 0.1-0. Previously the differential equations which arise were solved via a "locally produced" roll-your-own Runge-Kutta procedure. Now they are solved (in a more sophisticated manner) using the package deSolve. This increases the solution speed by a factor of about 7. There will be (minor, it is to be hoped) numerical differences in solutions produced from the same input.

Author(s)

Rolf Turner <r.turner@auckland.ac.nz> http://www.stat.auckland.ac.nz/~rolf

References

P. K. Banerjee and T. R. Turner (2012). A flexible model for the pricing of perishable assets. *Omega* **40**:5, 533–540. DOI https://doi.org/10.1016/j.omega.2011.10.001

Rolf Turner, Pradeep Banerjee and Rayomand Shahlori (2014). Optimal Asset Pricing. *Journal of Statistical Software* **58**:11, 1–25. DOI https://doi.org/10.18637/jss.v058.i11

See Also

xsolve(), plot.AssetPricing()

Examples

```
#
# In these examples "qmax" has been set equal to 5 which is
# an unrealistically low value for the total number of assets.
# This is done so as to reduce the time for package checking on CRAN.
#
S <- expression(exp(-kappa*x/(1+gamma*exp(-beta*t))))
attr(S,"parvec") <- c(kappa=10/1.5,gamma=9,beta=1)
lambda1 <- function(tt){
84*(1-tt)
}

# Optimal pricing policy assuming customers arrive singly:
X <- xsolve(S=S,lambda=lambda1,gprob=1,tmax=1,qmax=5)
lambda2 <- function(tt){
36*(1-tt)
}
# Expected values if the customers actually arrive in groups, using the
# (sub-optimal) pricing policy based on the (erroneous) assumption that
# they arrive singly.  Note that the two scenarios are ``comparable'' in
# that the expected total number of customers is 42 in each case.
V <- vsolve(S=S,lambda=lambda2,gprob=(5:1)/15,x=X$x,alpha=0.5)
```

---

xsolve *Optimal pricing policy*

Description

Determines (by solving a coupled system of differential equations) the optimal prices as functions of (residual) time for a number perishable assets. Prices may be discrete or continuous. For continuous prices, the price sensitivity function may either be a smooth (twice differentiable) function or a function which is piecewise linear in price.

Usage

```
xsolve(S, lambda, gprob=1, tmax=NULL, qmax, prices=NULL, nout=300,
       type="sip", alpha=NULL, salval=0, epsilon=NULL,
       method="lsoda",verbInt=0)
```

Arguments

S	An expression, or list of expressions, or a function or list of functions, specifying the price sensitivity functions S_j(x,t). See **Details**.

`lambda` A function (of residual time `t` — see `tmax`) or a positive constant, specifying the intensity of the (generally inhomogeneous) Poisson process of arrival times of groups of potential customers.

`gprob` A function (to calculate probabilities) or a numeric vector of probabilities determining the distribution of the size of an arriving group of customers.

`tmax` The maximum residual time; think of this as being the initial time at which the assets go on sale (with time *decreasing* to zero, at which point the value of each asset drops to the "salvage value" (`salval`), usually `0`). The system of differential equations is solved over the time interval [0,`tmax`]. See **Details**.

`qmax` The maximum number of assets available for sale, i.e. the number of assets available at the starting (residual) time `tmax`.

`prices` A numeric vector (with positive values) listing the possible prices at which items may be offered for sale in the discrete pricing scenario.

`nout` The number of points at which values of the solution are to be provided. These are taken to be equispaced on [0,`tmax`].

`type` Scalar character string taking one of the two values `"sip"` (singly indexed prices) and `"dip"` (doubly indexed prices). In the `"dip"` case the price of the asset which is quoted to the arriving group is allowed to depend upon the group size (as well as upon (residual) time as well as on the number of assets remaining for sale. In the `"sip"` case the quoted price does not depend upon group size.

`alpha` A numeric scalar between 0 and 1 specifying the probability that an arriving group of size `j > q` (where `q` is the number of assets remaining for sale) will *consider* purchasing (all of) these remaining assets. It is irrelevant (and defaults to 1 as a "place holder") if customers always arrive singly.

`salval` A (non-negative) numeric scalar specifying the "salvage value" of an asset — i.e. the quantity to which the value of an asset drops at residual time `t=0`. Usually `salval` is equal to 0.

`epsilon` A numeric scalar used in determining the optimal price in settings in which this involves maximizing over a discrete set. See **Details**. It defaults to `.Machine$double.eps^0.25` in the case of discrete prices and to `.Machine$double.eps^0.5` when the price sensitivity function is piecewise linear. It is ignored if the price sensitivity function is smooth.

`method` Character string specified the solution method to be used by the differential equation solver `ode()`. There is a fairly large number of possible methods. See `ode()` for details.

`verbInt` A scalar value which controls "verbosity". If `verbInt > 0` then a "progress report" is printed every `verbInt` *seconds* (roughly). That is if the current value of `Sys.time()` is greater than or equal to the value stored at the time of the last report, plus `verbInt` seconds, then a new "report" is printed out. If `verbInt` is less than or equal to 0 then the solution process runs "silently". See section **Progress Reports** for a bit more detail.

Details

If prices are modelled as being continuous, and if the price sensitivity function is differentiable, a coupled system of differential equations for the optimal prices is solved. If the prices are modelled

as being discrete or if the price sensitivity function is piecewise linear in price, then a coupled system of differential equations for the expected value of the stock is solved, with the optimal price being determined at each step by maximizing over an appropriate finite discrete set. These differential equations are solved by the `ode()` function from the `deSolve` package.

The components of the argument `S` should be provided as expressions when the price sensitivity functions are assumed to be smooth, and these should be amenable to differentiation with respect to `x` and `t` via the function `deriv3()`.

Note that in general the expression or expressions will depend upon a number of *parameters* as well as upon `x` and `t`. The values of these parameters are specified via an attribute or attributes. If `S` is a (single) expression it has (must have) an attribute called `parvec` which is a *named* vector of parameter values. If `S` is a list of expressions each entry of the list has (must have) such an attribute.

In the "piecewise linear" context `S` can be specified *only* as a single function. It is then assumed that the price sensitivity function for a group of size `j` is given by `S_j(x,t) = S(x,t)^j`. Such piecewise linear price sensitivity functions should be built using the function `buildS()`.

In the case of discrete prices the argument `S` must be a function or list of functions specifying the price sensitivity functions `S_j(x,t)`. These functions need only be defined for the prices listed in the `prices` argument.

If `S` is a single expression or function, then `S_j(x,t)` is taken to be this expression or function raised to the power `j`. If `S` is a list, then `S_j(x,t)` is taken to be its `j`-th entry.

In the case where argument `S` is a piecewise linear price sensitivity function, the argument `tmax` is, if not specified, taken to be the value of the corresponding attribute of `S`. In this setting, if `tmax` *is* specified it must be less than or equal to the corresponding attribute of `S`.

For discrete prices and for piecewise linear price sensitivity functions, determining the optimal price involves maximizing expected values over finite discrete sets. It can happen that the location of the maximum can make a sudden "jump" from one time step to the next, causing anomalous looking discontinuities in the optimal price functions. To avoid this, we check on the change in the expected value at each of the possible new prices as compared with that at the "previous" price.

If the maximal "improvement" in expected value is less than or equal to `epsilon` then the "new" price is set equal to the previous value. If the maximal improvement is greater than `epsilon` then those values of price, where the expected value is greater than the maximum value minus `epsilon`, are considered and the one which is closest to the previous price is chosen.

If `epsilon` is set equal to a value less than or equal to `0` then the smoothing strategy described above is dispensed with. In this case the maximum is taken to be the first of the (possibly) multiple maxima of the expected value.

Value

A list with components:

`x` The optimal pricing policy, chosen to maximize the expected value of the remaining assets at any given time; an object of class `flap` ("function list for asset pricing"). (In the case of discrete prices it also inherits from `pwc.flap` (`pwc` stands for "piecewise constant"), and if `type=="dip"` it also inherits from `di.flap`.) If `type=="sip"` it has the form of a list of functions `x_q(t)`, with `q` running from 1 to `qmax` If `type=="dip"` if has the form of a list of functions

x_qj(t) with q running from 1 to qmax and j running from 1 to min(q,jmax) where jmax is the maximum group size.

In the case of continuous prices these functions will be continuous functions created by splinefun(). In the case of discrete prices these functions will be piecewise constant (of class stepfun) created by stepfun().

Note that x has an *attribute* qmax specifying the maximum number of assets available for sale, i.e. the number of assets available at the starting (residual) time tmax. Of course if type=="sip" then this attribute is simply the length of x. Note that if type == "dip" then the entry x[[i]] is equal to the function x_qj(t) where i = (j-1)*(qmax - j/2) + q.

v — An object of class flap whose entries are (spline) functions v_q(t) specifying the (optimal) expected value of q assets at time t corresponding to the (optimal) pricing policy x.

vdot — An object of class flap whose entries are the derivatives (with respect to t) of the functions v_q(t) described above. The values of these derivatives are determined sequentially in the process of solving the system of differential equations for the optimal pricing policy.

Note

A substantial change was made to this package as of the change of version number from 0.0-11 to 0.1-0. Previously the differential equations which arise were solved via a "locally produced" roll-your-own Runge-Kutta procedure. Now they are solved (in a more sophisticated manner) using the package deSolve. This increases the solution speed by a factor of about 7. There will be (minor, it is to be hoped) numerical differences in solutions produced from the same input.

Progress Reports

The "progress reports" produced when verbInt > 0 consist of rough estimates of the percentage of [0,tmax] (the interval over which the differential equation is being solved) remaining to be covered. A rough estimate of the total elapsed time since the solution process started is also printed out.

Having "progress reports" printed out appears to have no (or at worst negligible) impact on computation time.'

Author(s)

Rolf Turner <r.turner@auckland.ac.nz> http://www.stat.auckland.ac.nz/~rolf

References

P. K. Banerjee, and T. R. Turner (2012). A flexible model for the pricing of perishable assets. *Omega* **40**:5, 533–540. DOI https://doi.org/10.1016/j.omega.2011.10.001

Rolf Turner, Pradeep Banerjee and Rayomand Shahlori (2014). Optimal Asset Pricing. *Journal of Statistical Software* **58**:11, 1–25. DOI https://doi.org/10.18637/jss.v058.i11

See Also

vsolve(), plot.AssetPricing(), buildS()

Examples

```
#
# In these examples "qmax" has been set equal to 5 which is
# an unrealistically low value for the total number of assets.
# This is done so as to reduce the time for package checking on CRAN.
#
# Smooth price sensitivity function.
S <- expression(exp(-kappa*x/(1+gamma*exp(-beta*t))))
attr(S,"parvec") <- c(kappa=10/1.5,gamma=9,beta=1)

# Optimal pricing policy assuming customers arrive singly:
lambda1 <- function(tt){
84*(1-tt)
}
X1 <- xsolve(S=S,lambda=lambda1,gprob=1,tmax=1,qmax=5,
             type="sip",verbInt=5)
# Optimal pricing policy assuming customers arrive in groups of
# size up to 5, with group size probabilities 1/3, 4/15, 1/5, 2/15,
# and 1/15 respectively.
lambda2 <- function(tt){
36*(1-tt)
}
X2 <- xsolve(S=S,lambda=lambda2,gprob=(5:1)/15,tmax=1,qmax=5,
             type="sip", alpha=0.5,verbInt=5)

# Note that the intensity functions lambda1() and lambda2() are
# such that the expected total number of customers is 42 in each case.

# Discrete prices:
lambda3 <- function(t){42}
S      <- function(x,t){
                e <- numeric(2)
                e[x==1] <- exp(-2*t)
                e[x==0.6] <- 1.0
                e
          }
X3 <- xsolve(S=S,lambda=lambda3,gprob=1,tmax=1,qmax=5,prices=c(1,0.6),
             type="sip",verbInt=5)

# Piecewise linear price sensitivity function.
#
# Take S as in the example for buildS.
# This takes a loonnngggg time; the procedure is slow
# in the piecewise linear setting.
## Not run:
l0 <- get("lambda",envir=environment(get("alpha",envir=environment(S))[[1]]))
lambda4 <- function(t){apply(l0(t),1,sum)}
X4 <- xsolve(S=S,lambda=lambda4,gprob=(5:1)/15,qmax=30,type="sip",
                  alpha=0.5,verbInt=20)

## End(Not run)
```

Index