Package ‘jrvFinance’

July 22, 2025

Title Basic Finance; NPV/IRR/Annuities/Bond-Pricing; Black Scholes

Version 1.4.3

Description Implements the basic financial analysis functions similar to (but not identical to) what is available in most spreadsheet software. This includes finding the IRR and NPV of regularly spaced cash flows and annuities. Bond pricing and YTM calculations are included. In addition, Black Scholes option pricing and Greeks are also provided.

Depends R (>= 3.0.0)

License GPL (>= 2)

Encoding UTF-8

VignetteBuilder knitr

Suggests knitr, markdown, rmarkdown

URL https://github.com/jrvarma/jrvFinance

BugReports https://github.com/jrvarma/jrvFinance/issues

RoxygenNote 7.1.1

NeedsCompilation no

Author Jayanth Varma [aut, cre]

Maintainer Jayanth Varma <jrvarma@iima.ac.in>

Repository CRAN

Date/Publication 2021-11-05 13:40:02 UTC

Contents

`jrvFinance-package` *Basic Finance: NPV/IRR/annuities, bond pricing, Black Scholes*

Description

This package implements the basic financial analysis functions similar to (but not identical to) what is available in most spreadsheet software. This includes finding the IRR, NPV and duration of possibly irregularly spaced cash flows and annuities. Bond pricing, YTM and duration calculations are included. Black Scholes option pricing, Greeks and implied volatility are also provided.

Details

Important functions include:

`npv`, `irr`, `duration`, `annuity.pv`, `bond.price`, `bond.yield`, `GenBS`, `GenBSImplied`

For more details, see the vignette

Author(s)

Prof. Jayanth R. Varma `<jrvarma@iima.ac.in>`

References

The 30/360 day count was converted from C++ code in the QuantLib library. The Newton Raphson solver was converted from C++ code in the Boost library annuity *Present Value of Annuity and Related Functions*

Description

Functions to compute present value and future value of annuities, to find instalment given the present value or future value. Can also find the rate or the number of periods given other parameters.

Usage

```
annuity.pv(
  rate,
  n.periods = Inf,
  instalment = 1,
  terminal.payment = 0,
  immediate.start = FALSE,
  cf.freq = 1,
  comp.freq = 1
)

annuity.fv(
  rate,
  n.periods = Inf,
  instalment = 1,
  terminal.payment = 0,
  immediate.start = FALSE,
  cf.freq = 1,
  comp.freq = 1
)

annuity.instalment(
  rate,
  n.periods = Inf,
  pv = if (missing(fv)) 1 else 0,
  fv = 0,
  terminal.payment = 0,
  immediate.start = FALSE,
  cf.freq = 1,
  comp.freq = 1
)

annuity.periods(
  rate,
  instalment = 1,
  pv = if (missing(fv)) 1 else 0,
  fv = 0,
  terminal.payment = 0,
```

```
  immediate.start = FALSE,
  cf.freq = 1,
  comp.freq = 1,
  round2int.digits = 3
)

annuity.rate(
  n.periods = Inf,
  instalment = 1,
  pv = if (missing(fv)) 1 else 0,
  fv = 0,
  terminal.payment = 0,
  immediate.start = FALSE,
  cf.freq = 1,
  comp.freq = 1
)

annuity.instalment.breakup(
  rate,
  n.periods = Inf,
  pv = 1,
  immediate.start = FALSE,
  cf.freq = 1,
  comp.freq = 1,
  period.no = 1
)
```

Arguments

`rate` The interest rate in decimal (0.10 or 10e-2 for 10%)

`n.periods` The number of periods in the annuity.

`instalment` The instalment (cash flow) per period.

`terminal.payment` Any cash flow at the end of the annuity. For example, a bullet repayment at maturity of the unamortized principal.

`immediate.start` Logical variable which is `TRUE` for immediate annuities (the first instalment is due immediately) and `FALSE` for deferred annuities (the first instalment is due at the end of the first period).

`cf.freq` Frequency of annuity payments: 1 for annual, 2 for semi-annual, 12 for monthly.

`comp.freq` Frequency of compounding of interest rates: 1 for annual, 2 for semi-annual, 12 for monthly, Inf for continuous compounding.

`pv` The present value of all the cash flows including the terminal payment.

`fv` The future value (at the end of the annuity) of all the cash flows including the terminal payment.

`round2int.digits`
: Used only in `annuity.periods`. If the computed number of periods is an integer when rounded to round2int.digits, then the rounded integer value is returned. With the default value of 3, 9.9996 is returned as 10, but 9.9994 and 9.39999999 are returned without any rounding.

`period.no`
: Used only in `annuity.instalment.breakup`. This is the period for which the instalment needs to be broken up into principal and interest parts.

Details

These functions are based on the Present Value relationship:

$$pv = fv \cdot df = terminal.payment \cdot df + \frac{instalment(1 - df)}{r}$$

where $df = (1 + r)^{-n.periods}$ is the $n.periods$ discount factor and r is the per period interest rate computed using rate, cf.freq and comp.freq.

It is intended that only one of pv or fv is used in any function call, but internally the functions use $pv + fv \cdot df$ as the LHS of the present value relationship under the assumption that only of the two is non zero.

The function `annuity.instalment.breakup` regards the annuity as a repayment of a loan equal to `pv` plus the present value of terminal.payment. The instalment paid in period period.no is broken up into the principal repayment (amortization) and interest components.

Value

For most functions, the return value is one of the arguments described above. For example `annuity.pv` returns `pv`. The only exception is `annuity.instalment.breakup`. This returns a list with the following components:

`opening.principal`
: The principal balance at the beginning of the period

`closing.principal`
: The principal balance at the end of the period

`interest.part`
: The portion of the instalment which represents interest

`principal.part`
: The portion of the instalment which represents principal repayment

Author(s)

Prof. Jayanth R. Varma <jrvarma@iima.ac.in>

bisection.root — *Find zero of a function by bracketing the zero and then using bisection.*

Description

Tries to find the zero of a function by using the bisection method (`uniroot`). To call `uniroot`, the zero must be bracketed by finding two points at which the function value has opposite signs. The main code in this function is a grid search to find such a pair of points. A geometric grid of points between `lower` and `guess` and also between `guess` and `upper`. This grid is searched for two neighbouring points across which the function changes sign. This brackets the root, and then we try to locate the root by calling `uniroot`

Usage

```
bisection.root(f, guess, lower, upper, nstep = 100, toler = 1e-06)
```

Arguments

`f`	The function whose zero is to be found. An R function object that takes one numeric argument and returns a numeric value. In an IRR application, this will be the NPV function. In an implied volatility application, the value will be the option price.
`guess`	The starting value (guess) from which the solver starts searching for the root. Must be positive.
`lower`	The lower end of the interval within which to search for the root. Must be positive.
`upper`	The upper end of the interval within which to search for the root. Must be positive.
`nstep`	The number of steps in the grid search to bracket the zero. See details.
`toler`	The criterion to determine whether a zero has been found. This is passed on to `uniroot`

Value

The root (or NA if the method fails)

Author(s)

Prof. Jayanth R. Varma

bonds *Bond pricing using yield to maturity.*

Description

bond.price computes the price given the yield to maturity bond.duration computes the duration given the yield to maturity bond.yield computes the yield to maturity given the price bond.prices, bond.durations and bond.yields are wrapper functions that use `mapply` to vectorize bond.price, bond.duration and bond.yield All arguments to bond.prices, bond.durations and bond.yields can be vectors. On the other hand, bond.price, bond.duration and bond.yield do not allow vectors Standard compounding and day count conventions are supported for all functions.

Usage

```
bond.price(
  settle,
  mature,
  coupon,
  freq = 2,
  yield,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  comp.freq = freq,
  redemption_value = 100
)

bond.yield(
  settle,
  mature,
  coupon,
  freq = 2,
  price,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  comp.freq = freq,
  redemption_value = 100
)

bond.duration(
  settle,
  mature,
  coupon,
  freq = 2,
  yield,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  modified = FALSE,
  comp.freq = freq,
  redemption_value = 100
)
```

```
bond.TCF(
  settle,
  mature,
  coupon,
  freq = 2,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  redemption_value = 100
)

bond.prices(
  settle,
  mature,
  coupon,
  freq = 2,
  yield,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  comp.freq = freq,
  redemption_value = 100
)

bond.yields(
  settle,
  mature,
  coupon,
  freq = 2,
  price,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  comp.freq = freq,
  redemption_value = 100
)

bond.durations(
  settle,
  mature,
  coupon,
  freq = 2,
  yield,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E"),
  modified = FALSE,
  comp.freq = freq,
  redemption_value = 100
)
```

Arguments

`settle` The settlement date for which the bond is traded. Can be a character string or any object that can be converted into date using `as.Date`.

mature	The maturity date of the bond. Can be a character string or any object that can be converted into date using as.Date
coupon	The coupon rate in decimal (0.10 or 10e-2 for 10%)
freq	The frequency of coupon payments: 1 for annual, 2 for semi-annual, 12 for monthly.
yield	The yield to maturity of the bond
convention	The daycount convention
comp.freq	The frequency of compounding of the bond yield: 1 for annual, 2 for semi-annual, 12 for monthly. Usually same as freq.
redemption_value	The principal amount that the bond will pay on maturity or call. Typically necessary when the bond is expected to be called at premium to par.
price	The clean price of the bond.
modified	A logical value used in duration. TRUE to return Modified Duration, FALSE otherwise

Value

bond.TCF returns a list of three components

t	A vector of cash flow dates in number of years
cf	A vector of cash flows
accrued	The accrued interest

Author(s)

Prof. Jayanth R. Varma <jrvarma@iima.ac.in>

---

coupons *Bond pricing using yield to maturity.*

Description

Convenience functions for finding coupon dates and number of coupons of a bond.

Usage

```
coupons.dates(settle, mature, freq = 2)

coupons.n(settle, mature, freq = 2)

coupons.next(settle, mature, freq = 2)

coupons.prev(settle, mature, freq = 2)
```

Arguments

settle	The settlement date for which the bond is traded. Can be a character string or any object that can be converted into date using as.Date.
mature	The maturity date of the bond. Can be a character string or any object that can be converted into date using as.Date
freq	The frequency of coupon payments: 1 for annual, 2 for semi-annual, 12 for monthly.

Author(s)

Prof. Jayanth R. Varma <jrvarma@iima.ac.in>

---

daycount — *Day count and year fraction for bond pricing*

Description

Implements 30/360, ACT/360, ACT/360 and 30/360E day count conventions.

Usage

```
yearFraction(
  d1,
  d2,
  r1,
  r2,
  freq = 2,
  convention = c("30/360", "ACT/ACT", "ACT/360", "30/360E")
)

daycount.actual(d1, d2, variant = c("bond"))

daycount.30.360(d1, d2, variant = c("US", "EU", "IT"))
```

Arguments

d1	The starting date of period for day counts
d2	The ending date of period for day counts
r1	The starting date of reference period for ACT/ACT day counts
r2	The ending date of reference period for ACT/ACT day counts
freq	The frequency of coupon payments: 1 for annual, 2 for semi-annual, 12 for monthly.
convention	The daycount convention
variant	Three variants of the 30/360 convention are implemented, but only one variant of ACT/ACT is currently implemented

Author(s)

Prof. Jayanth R. Varma <jrvarma@iima.ac.in>

References

The 30/360 day count was converted from C++ code in the QuantLib library

---

`duration` *Duration and Modified Duration*

Description

Computes Duration and Modified Duration for cash flows with different cash flow and compounding conventions. Cash flows need not be evenly spaced.

Usage

```
duration(
  cf,
  rate,
  cf.freq = 1,
  comp.freq = 1,
 cf.t = seq(from = ifelse(immediate.start, 0, 1/cf.freq), by = 1/cf.freq, along.with =
    cf),
  immediate.start = FALSE,
  modified = FALSE
)
```

Arguments

`cf`	Vector of cash flows
`rate`	The interest rate in decimal (0.10 or 10e-2 for 10%)
`cf.freq`	Frequency of annuity payments: 1 for annual, 2 for semi-annual, 12 for monthly.
`comp.freq`	Frequency of compounding of interest rates: 1 for annual, 2 for semi-annual, 12 for monthly, Inf for continuous compounding.
`cf.t`	Optional vector of timing (in years) of cash flows. If omitted regular sequence of years is assumed.
`immediate.start`	Logical variable which is `TRUE` when the first cash flows is at the beginning of the first period (for example, immediate annuities) and `FALSE` when the first cash flows is at the end of the first period (for example, deferred annuities)
`modified`	in function duration, `TRUE` if modified duration is desired. `FALSE` otherwise.

edate *Shift date by a number of months*

Description

Convenience function for finding the same date in different months. Used for example to find coupon dates of bonds given the maturity date. See `coupons`

Usage

```
edate(from, months = 1)
```

Arguments

`from`	starting date - a character string or any object that can be converted into date using `as.Date`.
`months`	Number of months (can be negative)

equiv.rate *Equivalent Rates under different Compounding Conventions*

Description

Converts an interest rate from one compounding convention to another (for example from semi-annual to monthly compounding or from annual to continuous compounding)

Usage

```
equiv.rate(rate, from.freq = 1, to.freq = 1)
```

Arguments

`rate`	The interest rate in decimal (0.10 or 10e-2 for 10%)
`from.freq`	Frequency of compounding of the given interest rate: 1 for annual, 2 for semi-annual, 12 for monthly, Inf for continuous compounding.
`to.freq`	Frequency of compounding to which the given interest rate is to be converted: 1 for annual, 2 for semi-annual, 12 for monthly, `Inf` for continuous compounding.

GenBS — *Generalized Black Scholes model for pricing vanilla European options*

Description

Compute values of call and put options as well as the Greeks - the sensitivities of the option price to various input arguments using the Generalized Black Scholes model. "Generalized" means that the asset can have a continuous dividend yield.

Usage

```
GenBS(s, X, r, Sigma, t, div_yield = 0)
```

Arguments

`s`	the spot price of the asset (the stock price for options on stocks)
`X`	the exercise or strike price of the option
`r`	the continuously compounded rate of interest in decimal (0.10 or 10e-2 for 10%) (use `equiv.rate` to convert to a continuously compounded rate)
`Sigma`	the volatility of the asset price in decimal (0.20 or 20e-2 for 20%)
`t`	the maturity of the option in years
`div_yield`	the continuously compounded dividend yield (0.05 or 5e-2 for 5%) (use `equiv.rate` to convert to a continuously compounded rate)

Details

The Generalized Black Scholes formula for call options is
$e^{-rt}(s\ e^{gt}\ Nd1 - X\ Nd2)$
where
$g = r - div\_yield$
$Nd1 = N(d1)$ and $Nd2 = N(d2)$
$d1 = \frac{log(s/X)+(g+Sigma^2/2)t}{Sigma\sqrt{t}}$
$d2 = d1 - Sigma\sqrt{t}$
N denotes the normal CDF (`pnorm`)
For put options, the formula is
$e^{-rt}(-s\ e^{gt}\ Nminusd1 + X\ Nminusd2)$
where
$Nminusd1 = N(-d1)$ and $Nminusd2 = N(-d2)$

Value

A list of the following elements

`call`	the value of a call option

`put`	the value of a put option
`Greeks`	a list of the following elements
`Greeks$callDelta`	the delta of a call option - the sensitivity to the spot price of the asset
`Greeks$putDelta`	the delta of a put option - the sensitivity to the spot price of the asset
`Greeks$callTheta`	the theta of a call option - the time decay of the option value with passage of time. Note that time is measured in years. To find a daily theta divided by 365.
`Greeks$putTheta`	the theta of a put option
`Greeks$Gamma`	the gamma of a call or put option - the second derivative with respect to the spot price or the sensitivity of delta to the spot price
`Greeks$Vega`	the vega of a call or put option - the sensitivity to the volatility
`Greeks$callRho`	the rho of a call option - the sensitivity to the interest rate
`Greeks$putRho`	the rho of a put option - the sensitivity to the interest rate
`extra`	a list of the following elements
`extra$d1`	the d1 of the Generalized Black Scholes formula
`extra$d2`	the d2 of the Generalized Black Scholes formula
`extra$Nd1`	is `pnorm`(d1)
`extra$Nd2`	is `pnorm`(d2)
`extra$Nminusd1`	is `pnorm`(-d1)
`extra$Nminusd2`	is `pnorm`(-d2)
`extra$callProb`	the (risk neutral) probability that the call will be exercised = `Nd2`
`extra$putProb`	the (risk neutral) probability that the put will be exercised = `Nminusd2`

`GenBSImplied` *Generalized Black Scholes model implied volatility*

Description

Find implied volatility given the option price using the generalized Black Scholes model. "Generalized" means that the asset can have a continuous dividend yield.

Usage

```
GenBSImplied(
  s,
  X,
  r,
  price,
  t,
```

```
    div_yield,
    PutOpt = FALSE,
    toler = 1e-06,
    max.iter = 100,
    convergence = 1e-08
  )
```

Arguments

`s`	the spot price of the asset (the stock price for options on stocks)
`X`	the exercise or strike price of the option
`r`	the continuously compounded rate of interest in decimal (0.10 or 10e-2 for 10%) (use `equiv.rate` to convert to a continuously compounded rate)
`price`	the price of the option
`t`	the maturity of the option in years
`div_yield`	the continuously compounded dividend yield (0.05 or 5e-2 for 5%) (use `equiv.rate` to convert to a continuously compounded rate)
`PutOpt`	`TRUE` for put options, `FALSE` for call options
`toler`	passed on to `newton.raphson.root` The implied volatility is regarded as correct if the solver is able to match the option price to within less than `toler`. Otherwise the function returns `NA`
`max.iter`	passed on to `newton.raphson.root`
`convergence`	passed on to `newton.raphson.root`

Details

`GenBSImplied` calls `newton.raphson.root` and if that fails `uniroot`

`irr` *Internal Rate of Return*

Description

Computes IRR (Internal Rate of Return) for cash flows with different cash flow and compounding conventions. Cash flows need not be evenly spaced.

Usage

```
irr(
  cf,
  interval = NULL,
  cf.freq = 1,
  comp.freq = 1,
  cf.t = seq(from = 0, by = 1/cf.freq, along.with = cf),
  r.guess = NULL,
```

```
  toler = 1e-06,
  convergence = 1e-08,
  max.iter = 100,
  method = c("default", "newton", "bisection")
)
```

Arguments

cf	Vector of cash flows
interval	the interval c(lower, upper) within which to search for the IRR
cf.freq	Frequency of annuity payments: 1 for annual, 2 for semi-annual, 12 for monthly.
comp.freq	Frequency of compounding of interest rates: 1 for annual, 2 for semi-annual, 12 for monthly, Inf for continuous compounding.
cf.t	Optional vector of timing (in years) of cash flows. If omitted regular sequence of years is assumed.
r.guess	the starting value (guess) from which the solver starts searching for the IRR
toler	the argument toler for irr.solve. The IRR is regarded as correct if abs(NPV) is less than toler. Otherwise the irr function returns NA
convergence	the argument convergence for irr.solve
max.iter	the argument max.iter for irr.solve
method	The root finding method to be used. The default is to try Newton-Raphson method (newton.raphson.root) and if that fails to try bisection (bisection.root). The other two choices (newton and bisection force only one of the methods to be tried.

irr.solve — *Solve for IRR (internal rate of return) or YTM (yield to maturity)*

Description

This function computes the internal rate of return at which the net present value equals zero. It requires as input a function that computes the net present value of a series of cash flows for a given interest rate as well as the derivative of the NPV with respect to the interest rate (10,000 times this derivative is the PVBP or DV01). In this package, irr.solve is primarily intended to be called by the irr and bond.yield functions. It is made available for those who want to find IRR of more complex instruments.

Usage

```
irr.solve(
  f,
  interval = NULL,
  r.guess = NULL,
  toler = 1e-06,
```

```
  convergence = 1e-08,
  max.iter = 100,
  method = c("default", "newton", "bisection")
)
```

Arguments

f	The function whose zero is to be found. An R function object that takes one numeric argument and returns a list of two components (value and gradient). In the IRR applications, these two components will be the NPV and its derivative
interval	The interval c(lower, upper) within which to search for the IRR
r.guess	The starting value (guess) from which the solver starts searching for the IRR
toler	The argument toler to newton.raphson.root. The IRR is regarded as correct if abs(NPV) is less than toler. Otherwise the irr.solve returns NA
convergence	The argument convergence to newton.raphson.root.
max.iter	The maximum number of iterations of the Newton-Raphson procedure
method	The root finding method to be used. The default is to try Newton-Raphson method (newton.raphson.root) and if that fails to try bisection (bisection.root). The other two choices (newton and bisection force only one of the methods to be tried.

Details

The function irr.solve is basically an interface to the general root finder newton.raphson.root. However, if newton.raphson.root fails, irr.solve makes an attempt to find the root using uniroot from the R stats package. Uniroot uses bisection and it requires the root to be bracketed (the function must be of opposite sign at the two end points - lower and upper).

Value

The function irr.solve returns NA if the IRR/YTM could not be found. Otherwise it returns the IRR/YTM. When NA is returned, a warning message is printed

Author(s)

Prof. Jayanth R. Varma <jrvarma@iima.ac.in>

---

newton.raphson.root *A Newton Raphson root finder: finds x such that f(x) = 0*

Description

The function newton.raphson.root is a general root finder which can find the zero of any function whose derivative is available. In this package, it is called by irr.solve and by GenBSImplied. It can be used in other situations as well - see the examples below.

Usage

```
newton.raphson.root(
  f,
  guess = 0,
  lower = -Inf,
  upper = Inf,
  max.iter = 100,
  toler = 1e-06,
  convergence = 1e-08
)
```

Arguments

`f`	The function whose zero is to be found. An R function object that takes one numeric argument and returns a list of two components (value and gradient). In an IRR application, these two components will be the NPV and the DV01/10000. In an implied volatility application, the components will be the option price and the vega. See also the examples below
`guess`	The starting value (guess) from which the solver starts searching for the IRR
`lower`	The lower end of the interval within which to search for the root
`upper`	The upper end of the interval within which to search for the root
`max.iter`	The maximum number of iterations of the Newton-Raphson procedure
`toler`	The criterion to determine whether a zero has been found. If the value of the function exceeds `toler` in absolute value, then `NA` is returned with a warning
`convergence`	The relative tolerance threshold used to determine whether the Newton-Raphson procedure has converged. The procedure terminates when the last step is less than `convergence` times the current estimate of the root. Convergence can take place to a non zero local minimum. This is checked using the `toler` criterion below

Value

The function returns `NA` under either of two conditions: (a) the procedure did not converge after `max.iter` iterations, or (b) the procedure converged but the function value is not zero within the limits of `toler` at this point. The second condition usually implies that the procedure has converged to a non zero local minimum from which there is no downhill gradient.

If the iterations converge to a genuine root (within the limits of `toler`), then it returns the root that was found.

References

The Newton Raphson solver was converted from C++ code in the Boost library npv *Net Present Value*

Description

Computes NPV (Net Present Value) for cash flows with different cash flow and compounding conventions. Cash flows need not be evenly spaced.

Usage

```
npv(
  cf,
  rate,
  cf.freq = 1,
  comp.freq = 1,
 cf.t = seq(from = if (immediate.start) 0 else 1/cf.freq, by = 1/cf.freq, along.with =
    cf),
  immediate.start = FALSE
)
```

Arguments

`cf`	Vector of cash flows
`rate`	The interest rate in decimal (0.10 or 10e-2 for 10%)
`cf.freq`	Frequency of annuity payments: 1 for annual, 2 for semi-annual, 12 for monthly.
`comp.freq`	Frequency of compounding of interest rates: 1 for annual, 2 for semi-annual, 12 for monthly, Inf for continuous compounding.
`cf.t`	Optional vector of timing (in years) of cash flows. If omitted regular sequence of years is assumed.
`immediate.start`	Logical variable which is `TRUE` when the first cash flows is at the beginning of the first period (for example, immediate annuities) and `FALSE` when the first cash flows is at the end of the first period (for example, deferred annuities)

Index