Dcf Documentation

Release 0.7 [4 - Beta]

sonntagsgesicht, based on a fork of Deutsche Postbank [pbrisk

Tuesday, 31 May 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Python library *dcf*

A Python library for generating discounted cashflows *(dcf)*. Typical banking business methods are provided like interpolation, compounding, discounting and fx.

1.2 Example Usage

```
>>> from dcf import ZeroRateCurve
```

```
>>> time_grid = [0, 2]
>>> rate_grid = [.03, .05]
```

```
>>> ZeroRateCurve(time_grid, rate_grid).get_zero_rate(0, 1)
0.04
```

```
>>> ZeroRateCurve(time_grid, rate_grid).get_discount_factor(0, 1)
0.9607894391523232
```

Or use datetime

```
>>> from datetime import date
```

```
>>> start = date(2013,1,1)
>>> mid = date(2014,1,1)
>>> end = date(2015,1,1)
```

```
>>> ZeroRateCurve([start, end], [.03, .05]).get_zero_rate(start, mid)
0.04
```

```
>>> ZeroRateCurve([start, end], [.03, .05]).get_discount_factor(start, mid)
0.9608157444936446
```

The framework works fine with native datetime but we recommend businessdate package for more convenient functionality to roll out date schedules.

```
>>> from businessdate import BusinessDate, BusinessSchedule
```

```
>>> today = BusinessDate(20201031)
>>> schedule = BusinessSchedule(today, today + "8q", step="1q")
>>> schedule
[BusinessDate(20201031), BusinessDate(20210131), BusinessDate(20210430),␣
↪BusinessDate(20210731), BusinessDate(20211031), BusinessDate(20220131),␣
↪BusinessDate(20220430), BusinessDate(20220731), BusinessDate(20221031)]
```

To build payment plans for, e.g. annuity loans, pick a plan function and generate an redemption amount list for paying back the loan notional amount.

```
>>> from dcf.plans import annuity, outstanding
```

```
>>> number_of_payments = 8
>>> interest_rate = 0.02
>>> notional = 1000.
```

```
>>> plan = annuity(number_of_payments, amount=notional, fixed_rate=interest_rate)
>>> plan
[116.50979913376267, 118.83999511643792, 121.21679501876667, 123.64113091914203, 126.
↪11395353752485, 128.63623260827535, 131.20895726044085, 133.83313640564967]
```

```
>>> sum(plan)
1000.0
```

```
>>> out = outstanding(plan, amount=notional)
>>> out
[1000.0, 883.4902008662373, 764.6502057497994, 643.4334107310327, 519.7922798118907,␣
↪393.6783262743659, 265.0420936660905, 133.83313640564967]
```

```
>>> compound = [o * interest_rate + p for o, p in zip(out, plan)]
>>> compound
[136.50979913376267, 136.50979913376267, 136.50979913376267, 136.50979913376267, 136.
↪50979913376267, 136.50979913376267, 136.50979913376267, 136.50979913376267]
```

Putting all together and feeding the plan into a *FixedCashFlowList* and the list of outstanding into a *RateCashflowList* gives the legs of a loan.

```
>>> from businessdate import BusinessDate, BusinessSchedule
>>> from dcf import FixedCashFlowList, RateCashFlowList
>>> from dcf.plans import amortize, outstanding
```

Again, build a date schedule.

```
>>> today = BusinessDate(20201031)
>>> schedule = BusinessSchedule(today, today + "8q", step="1q")
>>> start_date, payment_dates = schedule[0], schedule[1:]
```

Fixing the properties of the product and rolling out the payment plan and list of notional outstanding.

```
>>> number_of_payments = 8
>>> interest_rate = 0.01
>>> notional = 1000.
```

```
>>> plan = amortize(number_of_payments, amount=notional)
>>> out = outstanding(plan, amount=notional)
```

Finally, create for each leg a *CashFlowList*.

```
>>> principal = FixedCashFlowList([start_date], [-notional], origin=start_date)
>>> print(principal)
FixedCashFlowList([BusinessDate(20201031) ... BusinessDate(20201031)], [-1000.0 ... -
↪1000.0], origin=BusinessDate(20201031))
```

```
>>> redemption = FixedCashFlowList(payment_dates, plan, origin=start_date)
>>> print(redemption)
FixedCashFlowList([BusinessDate(20210131) ... BusinessDate(20221031)], [125.0 ... 125.
↪0], origin=BusinessDate(20201031))
```

```
>>> interest = RateCashFlowList(payment_dates, out, origin=start_date, fixed_
↪rate=interest_rate)
>>> print(interest)
RateCashFlowList([BusinessDate(20210131) ... BusinessDate(20221031)], [1000.0 ... 125.
↪0], origin=BusinessDate(20201031))
```

Add those legs to *CashFlowLegList* provides a smart container for valuation (*get_present_value()*).

```
>>> from dcf import CashFlowLegList, ZeroRateCurve, get_present_value
```

```
>>> loan = CashFlowLegList([principal, redemption, interest])
>>> curve = ZeroRateCurve([today, today + '2y'], [-.005, .01])
```

```
>>> get_present_value(cashflow_list=loan, discount_curve=curve, valuation_date=today)
4.935421637918839
```

Moreover, variable interest derived from float rates as given by a forward rate curve, e.g. a *CashRateCurve*, are possible, too.

```
>>> from dcf import CashRateCurve, RateCashFlowList
>>> from tabulate import tabulate
```

```
>>> fwd = CashRateCurve([today, today + '2y'], [-.005, .007])
>>> spread = .001
>>> float_interest = RateCashFlowList(payment_dates, out, origin=start_date, fixed_
↪rate=spread, forward_curve=fwd, pay_offset='2b', fixing_offset='2b')
```

```
>>> print(tabulate(float_interest.table, headers='firstrow'))
  cashflow  pay date      notional  start date    end date      year fraction   ␣
↪fixed rate    forward rate  fixing date    tenor
----------  ----------  ----------  ------------  ----------  ---------------  -------
↪-----  --------------  -------------  -------
 -0.996578  20210131          1000  20201029      20210128           0.249144        ␣
↪0.001    -0.005        20201027       3M
 -0.554077  20210430           875  20210128      20210428           0.246407        ␣
↪0.001    -0.00356986   20210126       3M
 -0.205991  20210731           750  20210428      20210729           0.251882        ␣
↪0.001    -0.00209041   20210426       3M
```

(continues on next page)

(continued from previous page)

```
  0.065699  20211031           625  20210729      20211028           0.249144  ␣
↪0.001    -0.000578082  20210727       3M
  0.238906  20220131           500  20211028      20220127           0.249144  ␣
↪0.001     0.000917808  20211026       3M
  0.318939  20220430           375  20220127      20220428           0.249144  ␣
↪0.001     0.0024137    20220125       3M
  0.305799  20220731           250  20220428      20220728           0.249144  ␣
↪0.001     0.00390959   20220426       3M
  0.199486  20221031           125  20220728      20221027           0.249144  ␣
↪0.001     0.00540548   20220726       3M
```

```
>>> get_present_value(cashflow_list=float_interest, discount_curve=curve, valuation_
↪date=today)
-0.641528888054065
```

1.3 Install

The latest stable version can always be installed or updated via pip:

```
$ pip install dcf
```

1.4 Development Version

The latest development version can be installed directly from GitHub:

```
$ pip install --upgrade git+https://github.com/sonntagsgesicht/dcf.git
```

1.5 Contributions

Issues and Pull Requests are always welcome.

1.6 License

Code and documentation are available according to the Apache Software License (see LICENSE).

CHAPTER

TWO

TUTORIAL

To start with import the package.

```
>>> from dcf import Curve, DateCurve, RateCurve
```

2.1 Interest Rate Curve Objects

2.1.1 Curves Types

Interest rate curves can be expressed in various ways. Each for different type of storing rate information and purpose.

There are four different types of interest rate curves:

> *dcf.curves.interestratecurve.ZeroRateCurve*, *dcf.curves.interestratecurve.DiscountFactorCurve*, *dcf.curves.interestratecurve.CashRateCurve* and *dcf.curves.interestratecurve.ShortRateCurve*.

They meet all the same interface, i.e. have the same properties and methods. They differ only in data which can be given for constructing the curves. This sets how rates are stored and interpolated. Moreover an large list of interpolation methods are provided.

As the names indicate, these curves take either

- zero (bond) rates,
- discount factors,
- forward cash interest rates or
- instantaneous interest rates aka. short rates

2.1.2 Getting Curve Values

From each class offers teh same methods to calculate each of those four types of rate by

> *dcf.curves.interestratecurve.InterestRateCurve.get_zero_rate()*, *dcf.curves.interestratecurve.InterestRateCurve.get_discount_factor()*, *dcf.curves.interestratecurve.InterestRateCurve.get_cash_rate()*, *dcf.curves.interestratecurve.InterestRateCurve.get_short_rate()*

2.1.3 Casting Curves

Even casting one type to another is as easy as

```
>>> from businessdate import BusinessDate
>>> from dcf import ZeroRateCurve, DiscountFactorCurve

>>> today = BusinessDate(20201031)
>>> date = today + '1m'

>>> zr_curve = ZeroRateCurve([today, today + '2y'], [-.005, .01])
>>> df_curve = DiscountFactorCurve(zr_curve)
>>> re_curve = ZeroRateCurve(df_curve)

>>> zr_curve(date), df_curve(date), re_curve(date)
(-0.004383561643835616, 1.0003601109552978, -0.004383561643827753)
```

2.2 Credit Curve Objects

Similar to *dcf.curves.interestratecurve.InterestRateCurve* *dcf.curves.creditcurve.CreditCurve* come in different (storage) types.

2.2.1 Curves Types

These are

dcf.curves.creditcurve.SurvivalProbabilityCurve, *dcf.curves.creditcurve.DefaultProbabilityCurve*, *dcf.curves.creditcurve.FlatIntensityCurve*, *dcf.curves.creditcurve.HazardRateCurve*, *dcf.curves.creditcurve.MarginalSurvivalProbabilityCurve*, *dcf.curves.creditcurve.MarginalDefaultProbabilityCurve*

2.2.2 Getting Curve Values

From each class offers teh same methods to calculate each of those four types of rate by

dcf.curves.creditcurve.CreditCurve.get_flat_intensity(), *dcf.curves.creditcurve.CreditCurve.get_hazard_rate()*, *dcf.curves.creditcurve.CreditCurve.get_survival_prob()*

2.2.3 Casting Curves

Even casting works the same way it does for interest rate curves.

2.3 Basic Curve Objects and Attributes

All of the mentioned curve classes inherit from these base classes.

2.3.1 Curve

Most fundamental is the *dcf.curves.curve.Curve* which sets the interpolation method. But note, even domain data (*x*-values), accessed by *dcf.curves.curve.Curve.domain*, are assume to be float.

The *x*-values can be shifted *left* or *right* by add in a *negative* or *positive* float via *dcf.curves.curve.Curve.shifted()*.

2.3.2 Operations

2.3.3 DateCurve

Different is *dcf.curves.curve.DateCurve*. Here are *x*-values given by date. But also easily casted

```
>>> from businessdate import BusinessDate
>>> from dcf import Curve, DateCurve

>>> today = BusinessDate(20200915)

>>> date_curve = DateCurve([today, today + '4y'], [0.1, 0.3])

>>> curve = Curve(date_curve)

>>> x =  date_curve.day_count(date_curve.origin, today + '2y')
>>> curve(x), date_curve(today + '2y')
(0.19993155373032168, 0.19993155373032168)
```

Already use was the domain property *dcf.curves.curve.Curve.domain* or *dcf.curves.curve.DateCurve.domain* which reveals the *x*-values of the curve.

As we have to measure distances by a day counting method (aka. year fraction) *dcf.curves.curve.DateCurve.day_count()*, which is mainly turning days into `float`.

See wikipedia for details and also businessdate for an implementation.

Moreover a date *dcf.curves.curve.DateCurve.origin* is marking the origin of the *x*-axis

Finally, many curves can be integrated as well as the derivative can be derived. Same works for *dcf.curves.curve.DateCurve*:

dcf.curves.curve.DateCurve.integrate(), *dcf.curves.curve.DateCurve.derivative()*

2.4 Cashflow Objects and Valuation

2.4.1 Payment Plans

```
>>> from businessdate import BusinessDate, BusinessSchedule
```

```
>>> today = BusinessDate(20201031)
```

```
>>> schedule = BusinessSchedule(today, today + "8q", step="1q")
>>> schedule
[BusinessDate(20201031), BusinessDate(20210131), BusinessDate(20210430),␣
↪BusinessDate(20210731), BusinessDate(20211031), BusinessDate(20220131),␣
↪BusinessDate(20220430), BusinessDate(20220731), BusinessDate(20221031)]
```

To build payment plans for, e.g. annuity loans, pick a plan function and generate an redemption amount list for paying back the loan notional amount.

```
>>> from dcf.plans import annuity, outstanding

>>> number_of_payments = 8
>>> interest_rate = 0.02
>>> notional = 1000.

>>> plan = annuity(number_of_payments, amount=notional, fixed_rate=interest_rate)
>>> plan
[116.50979913376267, 118.83999511643792, 121.21679501876667, 123.64113091914203, 126.
↪11395353752485, 128.63623260827535, 131.20895726044085, 133.83313640564967]


>>> sum(plan)
1000.0

>>> out = outstanding(plan, amount=notional)
>>> out
[1000.0, 883.4902008662373, 764.6502057497994, 643.4334107310327, 519.7922798118907,␣
↪393.6783262743659, 265.0420936660905, 133.83313640564967]

>>> compound = [o * interest_rate + p for o, p in zip(out, plan)]
>>> compound
[136.50979913376267, 136.50979913376267, 136.50979913376267, 136.50979913376267, 136.
↪50979913376267, 136.50979913376267, 136.50979913376267, 136.50979913376267]
```

2.4.2 CashFlowList Objects

Putting all together and feeding the plan into a *FixedCashFlowList and the list of outstanding into a `RateCashflowList* gives the legs of a loan.

```
>>> from businessdate import BusinessDate, BusinessSchedule
>>> from dcf.plans import amortize, outstanding
>>> from dcf import FixedCashFlowList, RateCashFlowList
```

Again, build a date schedule.

```
>>> today = BusinessDate(20201031)

>>> schedule = BusinessSchedule(today, today + "8q", step="1q")

>>> start_date, payment_dates = schedule[0], schedule[1:]
```

Fixing the properties of the product and rolling out the payment plan and list of notional outstanding.

```
>>> number_of_payments = 8
>>> interest_rate = 0.01
>>> notional = 1000.
```

(continues on next page)

(continued from previous page)

```
>>> plan = amortize(number_of_payments, amount=notional)
>>> out = outstanding(plan, amount=notional)
```

Finally, create for each leg a *dcf.cashflows.cashflow.CashFlowList*.

```
>>> principal = FixedCashFlowList([start_date], [-notional], origin=start_date)
>>> print(principal)
FixedCashFlowList([BusinessDate(20201031) ... BusinessDate(20201031)], [-1000.0 ... -
↪1000.0], origin=BusinessDate(20201031))

>>> redemption = FixedCashFlowList(payment_dates, plan, origin=start_date)
>>> print(redemption)
FixedCashFlowList([BusinessDate(20210131) ... BusinessDate(20221031)], [125.0 ... 125.
↪0], origin=BusinessDate(20201031))

>>> interest = RateCashFlowList(payment_dates, out, origin=start_date, fixed_
↪rate=interest_rate)
>>> print(interest)
RateCashFlowList([BusinessDate(20210131) ... BusinessDate(20221031)], [1000.0 ... 125.
↪0], origin=BusinessDate(20201031))
```

2.4.3 Valuation

Add those legs to *dcf.cashflows.cashflow.CashFlowLegList* provides a smart container for valuation (*dcf.pricer.get_present_value()*).

```
>>> from dcf import CashFlowLegList, ZeroRateCurve, get_present_value

>>> loan = CashFlowLegList([principal, redemption, interest])
>>> curve = ZeroRateCurve([today, today + '2y'], [-.005, .01])
>>> pv = get_present_value(cashflow_list=loan, discount_curve=curve, valuation_
↪date=today)
>>> pv
4.935421637918839
```

CHAPTER

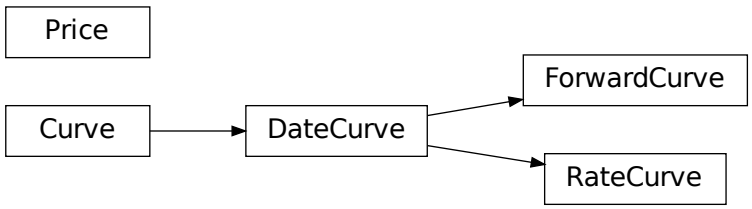
THREE

API DOCUMENTATION

3.1 Curve Objects

3.1.1 Basic Curves

Price	Price object for assets
Curve	Curve function object
DateCurve	Curve function object with dates as domain (points)
ForwardCurve	Forward price curve with yield extrapolation
RateCurve	Interest rate curve and credit curve



class dcf.curves.curve.**Price**(*value=0.0*, *origin=None*)

Bases: object

Price object for assets

Parameters

- **value** – price value
- **origin** – price date

```
>>> from businessdate import BusinessDate
>>> from dcf import Price
>>> p=Price(100, BusinessDate(20201212))
>>> p.value
100.0
>>> float(p)
100.0
>>> p
Price(100.000000; origin=BusinessDate(20201212))
```

property value

asset price value

property origin

asset price date

class dcf.curves.curve.**Curve**(*domain=()*, *data=()*, *interpolation=None*)

Bases: object

Curve function object

Parameters

- **domain** (*list(float)*) – source values $x_1 \dots x_n$
- **data** (*list(float)*) – target values $y_1 \dots y_n$
- **interpolation** (*function*) – (optional, default is defined on class level)

  Interpolation function γ such that $\gamma(x_i) = y_i$ for $i = 1 \dots n$.

  If **interpolation** is a string, the interpolation function is taken from class member dictionary *dcf.curves.curve.Curve.INTERPOLATIONS*.

  Interpolation functions γ can be constructed piecewise using via *dcf.interpolation.interpolation_scheme*.

Curve function object

$$f : \mathbb{R} \to \mathbb{R}, x \mapsto f(x) = y$$

build from finite point vectors x and y using piecewise various interpolation functions.

```
>>> from dcf import Curve
>>> c = Curve([0, 1, 2], [1, 2, 3])
```

get the grid of x values

```
>>> c.domain
[0, 1, 2]
```

get the grid of y values

```
>>> c(c.domain)
(1.0, 2.0, 3.0)
```

get a interpolated curve value

```
>>> c(1.5)
2.5
```

update existing values

```
>>> c[2] = 4
>>> c(c.domain)
(1.0, 2.0, 4.0)
```

add new points

```
>>> c[3] = 5
>>> c(c.domain)
(1.0, 2.0, 4.0, 5.0)
```

INTERPOLATIONS = {}

mapping (dict) of availiable interpolations additional to *dcf.interpolation*

property kwargs

returns constructor arguments as ordered dictionary

property domain

coordinates and date of given (not interpolated) x-values

property table

table of interpolated rates (pretty printable) given by `dcf.rate_table()`.

shifted(*delta=0.0*)

build curve object with shifted **domain** by **delta**

Parameters
delta – shift size

Returns
curve object with shifted **domain** by **delta**

class dcf.curves.curve.**DateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*)

Bases: *Curve*

Curve function object with dates as domain (points)

curve function object with dates as domain (points)

Parameters

- **domain** – squences of date points
- **data** – squence of curve values
- **interpolation** – interpolation function (see *dcf.curves.curve.Curve*)
- **origin** – inital origin of date points (used to calculate year fractions of poins in domain)
- **day_count** – day count function to derive year fractions from time periods

```
>>> from dcf import DateCurve
```

domain given as date/time measured in year fraction (float)

```
>>> domain = 0.5, 1.0, 1.5, 2.0
>>> data = 1, 2, 3, 4
```

```
>>> c = DateCurve(domain, data)
>>> c.domain
(0.5, 1.0, 1.5, 2.0)
```

```
>>> c(0.75)
1.5
```

domain given as date/time measured in dates (date)

```
>>> from datetime import date
```

```
>>> domain = date(2022, 8, 12), date(2023, 2, 12), date(2023, 8, 12), date(2024,␣
↪2, 12)
>>> data = 1, 2, 3, 4
```

```
>>> c = DateCurve(domain, data)
>>> c.domain
(datetime.date(2022, 8, 12), datetime.date(2023, 2, 12), datetime.date(2023, 8,␣
↪12), datetime.date(2024, 2, 12))
```

```
>>> c(date(2022, 11, 12))
1.5
```

domain given as date/time measured in dates (BusinessDate)

```
>>> from businessdate import BusinessDate
>>> t = BusinessDate(20220212)
```

```
>>> domain = tuple(t + p for p in ('6m', '12m', '18m', '24m'))
>>> data = 1, 2, 3, 4
```

```
>>> c = DateCurve(domain, data)
>>> c.domain
(BusinessDate(20220812), BusinessDate(20230212), BusinessDate(20230812),␣
↪BusinessDate(20240212))
```

```
>>> c(t + '9m')
1.5
```

DAY_COUNT = {}

mapping (dict) of availiable day count functions additional to *dcf.daycount*

property domain

domain of curve $t_1 \dots t_n$ as list of dates where curve values are given explicit

property origin

date of origin (date zero) as curve reference date for time calucations

day_count(*start*, *end=None*)

day count function to calculate a year fraction of time period

Parameters

- **start** – first date of period
- **end** – last date of period

Returns

(float) year fraction

to_curve()

deprecated method to cast to *dcf.curves.curve.Curve* object

integrate(*start*, *stop*)

integrates curve and returns results as annualized rates

Parameters

- **start** – lower integration boundary
- **stop** – upper integration boundary

Returns

(float) integral value$

If γ is this the curve. **integrate** returns

$$\int_a^b \gamma(t)\, dt$$

where a is **start** and b is **stop**.

if available **integrate** uses scipy.integrate.quad

derivative(*start*)

calculates numericaly the first derivative

Parameters
start – curve point to calcuate derivative at this point

Returns
(float) first derivative

If γ is this the curve **derivative** returns

$$\frac{d}{dt}\gamma(t)$$

where t is **start** but derived numericaly.

if available **derivative** uses scipy.misc.derivative

class dcf.curves.curve.**ForwardCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *yield_curve=0.0*)

Bases: *DateCurve*

Forward price curve with yield extrapolation

curve of future asset prices i.e. asset forward prices

Parameters

- **domain** – dates of given asset prices $t_1 \dots t_n$
- **data** – actual asset prices $p_{t_1} \dots p_{t_n}$
- **interpolation** – interpolation method for interpolating given asset prices
- **origin** – origin of curve
- **day_count** – day count method resp. function τ to calculate year fractions
- **yield_curve** – yield y to extrapolate by continous compounding

$$p_T = p_{t_n} \cdot \exp(y \cdot \tau(t_n, T))$$

or yield curve function γ_c to extrapolate by

$$p_T = p_{t_n} \cdot \gamma_c(T)/\gamma_c(t_n)$$

or interest rate curve c extrapolate by

$$p_T = p_{t_n} \cdot df_c^{-1}(t_n, T)$$

yield_curve

yield curve for extrapolation using discount factors

get_forward_price(*value_date*)

asset forward price at **value_date**

derived by interpolation on given forward prices and extrapolation by given discount_factor resp. yield curve

Parameters
value_date – future date of asset price

Returns
asset forward price at **value_date**

class dcf.curves.curve.**RateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *DateCurve*

Interest rate curve and credit curve

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

cast(*cast_type*, ***kwargs*)

deprecated method to cast a curve

property forward_tenor

tenor (time period) associated to the rates of the curve

property spread

spread curve to add spreads to curve

dcf.curves.curve.**rate_table**(*curve*, *x_grid=None*, *y_grid=None*)

table of calculated rates

Parameters

- **curve** – function f
- **x_grid** – vertical date axis $x_0, \dots, x_m$
- **y_grid** – horizontal period axis $y_1, \dots, y_n$ (implicitly added a non-period $y_0 = 0$)

Returns
list(list(float)) matrix $T = (t_{i,j})$ with $t_{i,j} = f(x_i + y_j)$ if $x_i + y_j < x_{i+1}$.

```
>>> from tabulate import tabulate
>>> from dcf import Curve, rate_table
>>> curve = Curve([1, 4], [0, 1])
>>> table = rate_table(curve, x_grid=(0, 1, 2, 3, 4, 5), y_grid=(.0, .25, .5, .
↪75))
>>> print(tabulate(table, headers='firstrow', floatfmt='.4f'))
       0.0    0.25     0.5    0.75
--  ------  ------  ------  ------
 0  0.0000  0.0000  0.0000  0.0000
```

(continues on next page)

(continued from previous page)

```
 1  0.0000  0.0833  0.1667  0.2500
 2  0.3333  0.4167  0.5000  0.5833
 3  0.6667  0.7500  0.8333  0.9167
 4  1.0000  1.0000  1.0000  1.0000
 5  1.0000  1.0000  1.0000  1.0000
```

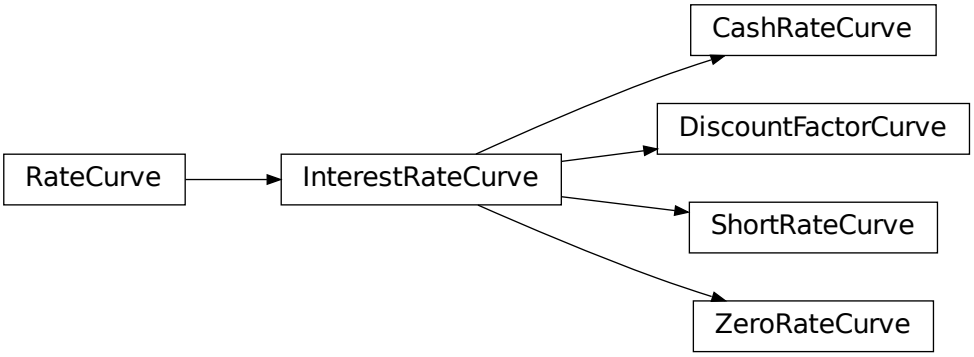
```
>>> from businessdate import BusinessDate, BusinessPeriod
>>> from dcf import ZeroRateCurve
```

```
>>> term = '1m', '3m', '6m', '1y', '2y', '5y',
>>> rates = -0.008, -0.0057, -0.0053, -0.0036, -0.0010, 0.0014,
>>> today = BusinessDate(20211201)
>>> tenor = BusinessPeriod('1m')
>>> dates = [today + t for t in term]
>>> f = ZeroRateCurve(dates, rates, origin=today, forward_tenor=tenor)
```

```
>>> print(tabulate(f.table, headers='firstrow', floatfmt=".4f", tablefmt='latex
↪'))
\begin{tabular}{lrrrrrrr}
\hline
          &      0D &      1M &      2M &      3M &      6M &      1Y &     2Y \\
\hline
 20211201 & -0.0080 &         &         &         &         &         &        \\
 20220101 & -0.0080 & -0.0068 &         &         &         &         &        \\
 20220301 & -0.0057 & -0.0056 & -0.0054 &         &         &         &        \\
 20220601 & -0.0053 & -0.0050 & -0.0047 & -0.0044 &         &         &        \\
 20221201 & -0.0036 & -0.0034 & -0.0032 & -0.0030 & -0.0023 &         &        \\
 20231201 & -0.0010 & -0.0009 & -0.0009 & -0.0008 & -0.0006 & -0.0002 & 0.0006 \\
 20261201 &  0.0014 &  0.0014 &  0.0014 &  0.0014 &  0.0014 &  0.0014 & 0.0014 \\
\hline
\end{tabular}
```

3.1.2 Interest Rate Curves

InterestRateCurve	Base class of interest rate curve classes
ZeroRateCurve	Interest rate curve storing and interpolating data as zero rates
DiscountFactorCurve	Interest rate curve storing and interpolating data as discount factor
CashRateCurve	Interest rate curve storing and interpolating data as cash rate
ShortRateCurve	Interest rate curve storing and interpolating data as short rate



class dcf.curves.interestratecurve.**InterestRateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *RateCurve*

Base class of interest rate curve classes

All interest rate curves share the same four fundamental methodological methods

- *dcf.curves.interestratecurve.InterestRateCurve.get_discount_factor()*
- *dcf.curves.interestratecurve.InterestRateCurve.get_zero_rate()*
- *dcf.curves.interestratecurve.InterestRateCurve.get_cash_rate()*
- *dcf.curves.interestratecurve.InterestRateCurve.get_short_rate()*
- *dcf.curves.interestratecurve.InterestRateCurve.get_swap_annuity()*

All subclasses differ only in data types for storage and interpolation.

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

get_discount_factor(*start*, *stop=None*)

discounting factor for future cashflows

Parameters

- **start** – date t_0 to discount to

- **stop** – date t_1 for discounting from (optional, if not given t_0 will be **origin** and t_1 by **start**)

Returns
discounting factor $df(t_0, t_1)$

Assuming a constant bank account interest rate r over time and interest rate compounding a bank account of $B_0 = 1$ at time t_0 will be some value B_1 at time t_1.

For continuous compounding $B_1 = B_0 * \exp(r \cdot (t_1 - t_0))$, for more concepts of compounding see *dcf.compounding*.

Since B_1 is equivalent to the value of B_0 at time t_1, B_0/B_1 can be understood to as the price at time t_0 of a bank account of 1 at t_1.

In general, discount factor $df(t_0, t_1) = B_0/B_1$ are used to give the price or present value $v_0(CF)$ at time t_0 of any cashflow CF at time t_1 by

$$v_0(CF) = df(t_0, t_1) \cdot CF.$$

This concept relates to the zero bond yields *dcf.curves.interestratecurve.InterestRateCurve.get_zero_rate()*.

get_zero_rate(*start*, *stop=None*)

curve of zero rates, i.e. yields of zero cupon bonds

Parameters

- **start** – zero bond start date t_0
- **stop** – zero bond end date t_1

Returns
zero bond rate $z(t_0, t_1)$

Assume a current price is $P(t_0, t_1)$ at time t_0 of a zero cupon bond P paying 1 at maturity t_1 without any interest or cupons.

Such zero bond prices are used to give the price or present value $v_0(CF)$ at time t_0 of any cashflow CF at time t_1 by

$$v_0(CF) = P(t_0, t_1) \cdot CF = \exp(-z(t_1 - t_0) \cdot \tau(t_1 - t_0)) \cdot CF$$

where τ is the day count method to calculate the year fraction of the interest accrual period form t_i to t_{i+1} given by *dcf.curves.curve.DateCurve.day_count()*.

Note, this concept relates to the discount factor $df(t_0, t_1)$ of *dcf.curves.interestratecurve.InterestRateCurve.get_discount_factor()* by

$$df(t_0, t_1) = \exp(-z(t_1 - t_0) \cdot \tau(t_1 - t_0)).$$

Note, this concept relates to short rates $df(t_0, t_1)$ of *dcf.curves.interestratecurve.InterestRateCurve.get_short_rate()* by

$$z(t_0, t_1)(t_1 - t_0) = \int_{t_0}^{t_1} r(t)dt.$$

get_short_rate(*start*)

constant interpolated short rate derived from zero rate

Parameters
start (*date*) – point in time t of short rate

Returns
short rate r_t at given point in time

Calculation assumes a zero rate derived from a interpolated short rate, i.e.

Let $r_t = r(t)$ be the short rate on given time grid $t_0, t_1, \ldots, t_n$ and let $z(s,t)$ be the zero rate from s to t with $s, t \in \{t_0, t_1, \ldots, t_n\}$.

Hence,

$$\int_s^t r(\tau)d\tau = \int_s^t c_s d\tau = \Big[c_s\tau\Big]_s^t = c_s(s-t)$$

and so

$$c_s = z(s,t).$$

See also *dcf.curves.interestratecurve.InterestRateCurve.get_zero_rate()*.

get_cash_rate(*start*, *stop=None*, *step=None*)

interbank cash lending rate

Parameters

- **start** – start date of cash lending
- **stop** – end date of cash lending (optional; default **start** + **step**)
- **step** – period length of cash lending (optional; by default **step** is taken from *dcf.curves.curve.RateCurve.forward_tenor*)

Returns
simple compounded interest (forward) rate f

Let **start** be t_0. If **step** and **stop** are given as τ and t_1 then **start** + **step** = **stop** must meet such that $t_0 + \tau = t_1$ in

$$f(t_0, t_1) = \frac{1}{\tau}\big(\frac{1}{df(t_0, t_1)} - 1\big).$$

Due to the benchmark reform most classical cash rates as the *LIBOR* rates have been replaced by overnight rates, e.g. *SOFR*, *SONIA* etc. Derived from future prediictions of overnight rates (aka *short term* rates) long term rates with tenors of $1m$, $3m$, $6m$ and $12m$ are published, too.

For classical term rates see LIBOR and EURIBOR, for overnight rates see SOFR, ESTR, SONIA and SARON as well as TONAR.

get_swap_annuity(*date_list*)

swap annuity as the accrual period weighted sum of discount factors

Parameters
date_list – list of period $t_0, \ldots t_n$

Returns
swap annuity $A(t_0, \ldots, t_n)$

As

$$A(t_0, \ldots, t_n) = \sum_{i=1}^{n} df(0, t_i)\tau(t_i, t_{i+1})$$

with

- 0 given by *dcf.curves.curve.DateCurve.origin*
- *df* discount factor given by *dcf.curves.interestratecurve.InterestRateCurve.get_discount_factor()*
- τ day count method to calculate the year fraction of the interest accrual period form t_i to t_{i+1} given by *dcf.curves.curve.DateCurve.day_count()*

class dcf.curves.interestratecurve.**ZeroRateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *InterestRateCurve*

Interest rate curve storing and interpolating data as zero rates

$$z(t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \ldots t_n$ or a curve object C
- **data** – either curve values $y_1 \ldots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.interestratecurve.**DiscountFactorCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *InterestRateCurve*

Interest rate curve storing and interpolating data as discount factor

$$df(t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \ldots t_n$ or a curve object C
- **data** – either curve values $y_1 \ldots y_n$ or a curve object C

- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s,t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.interestratecurve.**CashRateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *InterestRateCurve*

Interest rate curve storing and interpolating data as cash rate

$$f(t, t+\tau^*) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s,t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.interestratecurve.**ShortRateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *InterestRateCurve*

Interest rate curve storing and interpolating data as short rate

$$r(t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme

- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

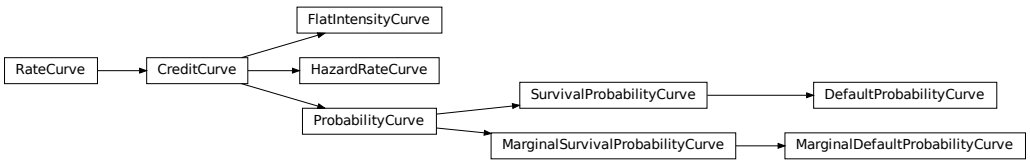
If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

3.1.3 Credit Curves

CreditCurve	Base class of credit curve classes
SurvivalProbabilityCurve	Interest rate curve storing and interpolating data as discount factor
FlatIntensityCurve	Credit curve storing and interpolating data as intensities
HazardRateCurve	Credit curve storing and interpolating data as hazard rate
MarginalSurvivalProbabilityCurve	Credit curve storing and interpolating data as intensities
MarginalDefaultProbabilityCurve	Credit curve storing and interpolating data as marginal default probability



class dcf.curves.creditcurve.**CreditCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *RateCurve*

Base class of credit curve classes

All credit curves share the same three fundamental methodological methods

- *dcf.curves.creditcurve.CreditCurve.get_survival_prob()*
- *dcf.curves.creditcurve.CreditCurve.get_flat_intensity()*
- *dcf.curves.creditcurve.CreditCurve.get_hazard_rate()*

All subclasses differ only in data types for storage and interpolation.

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C

- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

get_survival_prob(*start*, *stop=None*)

survival probability of credit curve

Parameters

- **start** – start point in time t_0 of period
- **stop** – end point t_1 of period (optional, if not given t_0 will be **origin** and t_1 taken from **start**)

Returns

survival probability $sv(t_0, t_1)$ for period t_0 to t_1

Assume an uncertain event χ, e.g. occurrence of a credit default event such as a loan borrower failing to fulfill the obligation to pay back interest or redemption.

Let ι_χ be the point in time when the event χ happens.

Then the survival probability $sv(t_0, t_1)$ is the probability of not occurring χ until t_1 if χ didn't happen until t_0, i.e.

$$sv(t_0, t_1) = 1 - P(t_0 < \iota_\chi \leq t_1)$$

- similar to *dcf.curves.interestratecurve.InterestRateCurve.get_discount_factor()*

get_flat_intensity(*start*, *stop=None*)

intensity value of credit curve

Parameters

- **start** – start point in time t_0 of intensity
- **stop** – end point t_1 of intensity (optional, if not given t_0 will be **origin** and t_1 taken from **start**)

Returns

intensity $\lambda(t_0, t_1)$

The intensity $\lambda(t_0, t_1)$ relates to survival probabilities by

$$sv(t_0, t_1) = exp(-\lambda(t_0, t_1) \cdot \tau(t_0, t_1)).$$

- similar to *dcf.curves.interestratecurve.InterestRateCurve.get_zero_rate()*

get_hazard_rate(*start*)

hazard rate of credit curve

Parameters
start – point in time t of hazard rate

Returns
hazard rate $hz(t)$

The hazard rate $hz(t)$ relates to intensities by

$$\lambda(t_0, t_1) = \int_{t_0}^{t_1} hz(t)\, dt.$$

- similar to *dcf.curves.interestratecurve.InterestRateCurve.get_short_rate()*

class dcf.curves.creditcurve.**ProbabilityCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *CreditCurve*

base class of probability based credit curve classes

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.creditcurve.**SurvivalProbabilityCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *ProbabilityCurve*

Interest rate curve storing and interpolating data as discount factor

$$sv(0, t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$

- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.creditcurve.**DefaultProbabilityCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *SurvivalProbabilityCurve*

Credit curve storing and interpolating data as default probability

$$pd(0,t) = 1 - sv(0,t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s,t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.creditcurve.**FlatIntensityCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *CreditCurve*

Credit curve storing and interpolating data as intensities

$$\lambda(t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s,t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.creditcurve.**HazardRateCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *CreditCurve*

Credit curve storing and interpolating data as hazard rate

$$hz(t) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.creditcurve.**MarginalSurvivalProbabilityCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *ProbabilityCurve*

Credit curve storing and interpolating data as intensities

$$sv(t, t + \tau^*) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

class dcf.curves.creditcurve.**MarginalDefaultProbabilityCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *forward_tenor=None*)

Bases: *MarginalSurvivalProbabilityCurve*

Credit curve storing and interpolating data as marginal default probability

$$pd(t, t+\tau^*) = 1 - sv(t, t+\tau^*) = y_t$$

Parameters

- **domain** – either curve points $t_1 \dots t_n$ or a curve object C
- **data** – either curve values $y_1 \dots y_n$ or a curve object C
- **interpolation** – (optional) interpolation scheme
- **origin** – (optional) curve points origin t_0
- **day_count** – (optional) day count convention function $\tau(s, t)$
- **forward_tenor** – (optional) forward rate tenor period τ^*

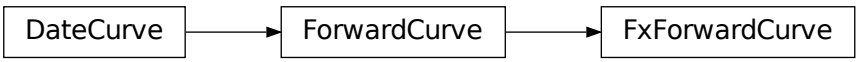
If **data** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given by **domain**.

If **domain** is a *dcf.curves.curve.RateCurve* instance C, it is casted to this new class type with domain grid given **domain** property of C.

Further arguments **interpolation**, **origin**, **day_count**, **forward_tenor** will replace the ones given by C if not given explictly.

3.1.4 Fx Curve

FxForwardCurve	fx rate curve for currency pair



class dcf.curves.fx.**FxRate**(*value=0.0*, *origin=None*)

Bases: *Price*

price object for foreign currency exchange rates

Parameters

- **value** – price value

- **origin** – price date

```
>>> from businessdate import BusinessDate
>>> from dcf import Price
>>> p=Price(100, BusinessDate(20201212))
>>> p.value
100.0
>>> float(p)
100.0
>>> p
Price(100.000000; origin=BusinessDate(20201212))
```

property origin

asset price date

property value

asset price value

class dcf.curves.fx.**FxForwardCurve**(*domain=()*, *data=()*, *interpolation=None*, *origin=None*, *day_count=None*, *domestic_curve=None*, *foreign_curve=None*)

Bases: *ForwardCurve*

fx rate curve for currency pair

get_forward_price(*value_date*)

asset forward price at **value_date**

derived by interpolation on given forward prices and extrapolation by given discount_factor resp. yield curve

Parameters
value_date – future date of asset price

Returns
asset forward price at **value_date**

3.2 Cashflow Objects

3.2.1 Build Functions

dcf.plans.**same**(*num*, *amount=1.0*)

all same payment plan

Parameters

- **num** – number of payments n
- **amount** – amount of each payment N

Returns
list(float) payment plan X_i for $i = 1 \dots n$

Payment plan with

$$X_i = N \text{ for all } i = 1 \dots n$$

dcf.plans.**bullet**(*num*, *amount=1.0*)

bullet payment plan

Parameters

- **num** – number of payments n

- **amount** – amount of last bullet payment N

Returns
list(float) payment plan X_i for $i = 1 \dots n$

Payment plan with

$$X_i = N \text{ for } i = n \text{ else } 0$$

dcf.plans.**amortize**(*num*, *amount=1.0*)

linear amortize payment plan

Parameters

- **num** – number of payments n
- **amount** – amount of total sum of payment N

Returns
list(float) payment plan X_i for $i = 1 \dots n$

Payment plan with

$$X_i = N/n \text{ for } i = 1 \dots n$$

dcf.plans.**annuity**(*num*, *amount=1.0*, *fixed_rate=0.01*)

fixed rate annuity payment plan

Parameters

- **num** – number of payments n
- **amount** – amount of total sum of payment N
- **fixed_rate** – amortization rate r

Returns
list(float) payment plan X_i for $i = 1 \dots n$

Payment plan

$$X_i = \frac{r}{(1+r)^{n-i}} \cdot N \text{ for } i = 1 \dots n$$

dcf.plans.**consumer**(*num*, *amount=1.0*, *fixed_rate=0.01*)

consumer loan annuity payment plan

Parameters

- **num** – number of payments n
- **amount** – amount of payment total N
- **fixed_rate** – amortization rate r

Returns
list(float) payment plan X_i for $i = 1 \dots n$

Actutal payment plan total $T = N(1 + n \cdot r)$ such that

$$X_i = T/n$$

dcf.plans.**outstanding**(*plan*, *amount=1.0*, *sign=False*)

sums up plans to remaining oustanding anmount

Parameters

- **plan** – payment plan X_i
- **amount** – inital amount N
- **sign** – σ sign of plan payments (optional, default: **-1**)

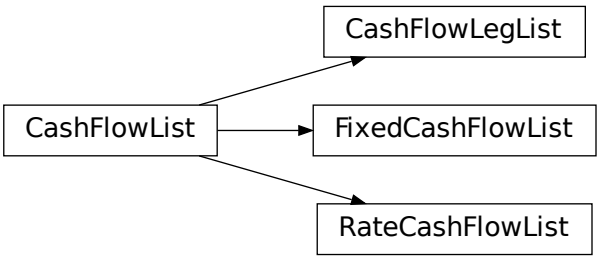
Returns

list(float) outstanding plan

Adds or substracts payment plan payments X_i from inital amount N such that

$$O_i = N + \sigma \cdot \sum_{k=1}^{i-1} X_i$$

3.2.2 Cashflow Objects



class dcf.cashflows.cashflow.**CashFlowList**(*payment_date_list=()*, *amount_list=()*, *origin=None*)

Bases: `object`

basic cashflow list object

Parameters

- **domain** – list of cashflow dates
- **data** – list of cashflow amounts
- **origin** – origin of object, i.e. start date of the cashflow list as a product

Basicly *dcf.cashflows.cashflow.CashFlowList* works like a read-only dictionary with payment dates as keys.

And the *dcf.cashflows.cashflow.CashFlowList.domain* property holds the payment date list.

```
>>> from dcf import CashFlowList
>>> cf_list = CashFlowList([0, 1], [-100., 100.])
>>> cf_list.domain
(0, 1)
```

In order to get cashflows

```
>>> cf_list[0]
-100.0
>>> cf_list[cf_list.domain]
(-100.0, 100.0)
```

This works even for dates without cashflow

```
>>> cf_list[-1, 0 , 1, 2]
(0.0, -100.0, 100.0, 0.0)
```

property table

cashflow details as list of tuples

property domain

payment date list

property origin

cashflow list start date

property kwargs

returns constructor arguments as ordered dictionary (under construction)

payoff(*date*)

dictionary of payoffs with pay_date keys

class dcf.cashflows.cashflow.**CashFlowLegList**(*legs*)

Bases: *CashFlowList*

MultiCashFlowList

container class for CashFlowList

Parameters

legs – list of *dcf.cashflows.cashflow.CashFlowList*

property legs

list of *dcf.cashflows.cashflow.CashFlowList*

class dcf.cashflows.cashflow.**FixedCashFlowList**(*payment_date_list*, *amount_list=1.0*, *origin=None*)

Bases: *CashFlowList*

basic cashflow list object

Parameters

- **payment_date_list** – list of cashflow payment dates
- **amount_list** – list of cashflow amounts
- **origin** – origin of object, i.e. start date of the cashflow list as a product

class dcf.cashflows.cashflow.**RateCashFlowList**(*payment_date_list*, *amount_list=1.0*, *origin=None*, *day_count=None*, *fixing_offset=None*, *pay_offset=None*, *fixed_rate=0.0*, *forward_curve=None*)

Bases: *CashFlowList*

list of cashflows by interest rate payments

list of interest rate cashflows

Parameters

- **payment_date_list** – pay dates, assuming that pay dates agree with end dates of interest accrued period

- **amount_list** – notional amounts
- **origin** – start date of first interest accrued period
- **day_count** – day count convention
- **fixing_offset** – time difference between interest rate fixing date and interest period payment date
- **pay_offset** – time difference between interest period end date and interest payment date
- **fixed_rate** – agreed fixed rate
- **forward_curve** – interest rate curve for forward estimation

Let t_0 be the list **origin** and $t_i \; i = 1, \dots n$ the **payment_date_list** with $N_i \; i = 1, \dots n$ the notional **amount_list**.

Moreover, let τ be the **day_count** function, c the **fixed_rate** and f the **forward_curve**.

Then, the rate cashflow cf_i payed at time t_i will be with $s_i = t_{i-1} - \delta$, $e_i = t_i - \delta$ as well as $d_i = s_i - \epsilon$ for **pay_offset** δ and **fixing_offset** ϵ,

$$cf_i = N_i \cdot \tau(s_i, e_i) \cdot (c + f(d_i)).$$

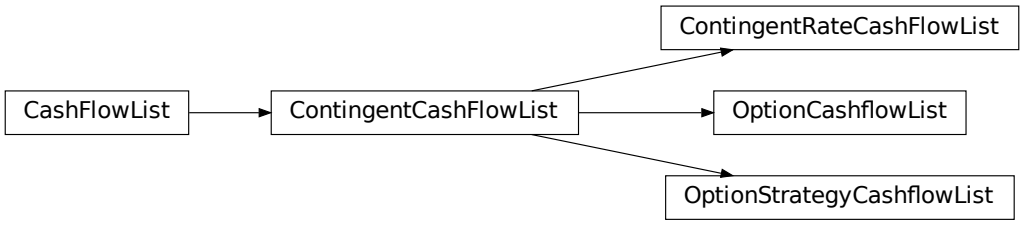
Note, the **pay_offset** δ is not applied in case of the first cashflow, then $s_1 = t_0$.

forward_curve

cashflow forward curve to derive float rates f

property fixed_rate

3.2.3 Contingent Cashflow Objects (Options)



class dcf.cashflows.contingent.**ContingentCashFlowList**(*payment_date_list*, *payoff_list=None*, *origin=None*, *payoff_model=None*)

Bases: *CashFlowList*

list of contingent cashflows

generic cashflow list of expected contingent cashflows i.e. non-deterministc cashflows like option payoffs.

Parameters

- **payment_date_list** – pay dates, assuming that pay dates agree with end dates of interest accrued period
- **payoff_list** – list of payoffs
- **origin** – start date of first interest accrued period
- **payoff_model** – payoff model to derive the expected payoff

Since expectation depends on probabilities an approbiate **payoff_model** m to estimate expectations has to be supplied as argument to the list and applied - again as argument - to payoffs.

Therefor any item f_i in **payoff_list** has to be either a **int** pr **float** or callable with optional argument of a **payoff_model** and will return the expected cashflow amount as float value depending on the state given by the **payoff_model**.

$$f_i(m) = E\big[f_i \mid m\big]$$

This non-sense use case demonstrates the pattern of evaluating payoffs. For more details who to use *dcf.cashflows.contingent.ContingentCashFlowList* see *dcf.cashflows.contingent.OptionCashflowList*, *dcf.cashflows.contingent.OptionStrategyCashflowList* or *dcf.cashflows.contingent.ContingentRateCashFlowList*.

```
>>> from dcf import ContingentCashFlowList
>>> p = lambda x: x*x
>>> c = ContingentCashFlowList([1,2], [p, p], payoff_model=4)
>>> c[c.domain]
[16, 16]
>>> c.payoff_model = 2
>>> c[c.domain]
[4, 4]
```

class dcf.cashflows.contingent.**OptionCashflowList**(*payment_date_list*, *amount_list=1.0*, *strike_list=()*, *is_put_list=False*, *fixing_offset=None*, *pay_offset=None*, *origin=None*, *payoff_model=None*)

Bases: *ContingentCashFlowList*

list of option cashflows

list of European option payoffs

Parameters

- **payment_date_list** – list of cashflow payment dates t_k
- **amount_list** – list of option notional amounts N_k
- **strike_list** – list of option strike prices K_k
- **is_put_list** – list of boolean flags indicating if options are put options (optional: default is **False**)
- **fixing_offset** – offset δ between underlying fixing date and cashflow end date
- **pay_offset** – offset ϵ between cashflow end date and payment date
- **origin** – origin of object, i.e. start date of the cashflow list as a product
- **payoff_model** – payoff model to derive the expected payoff

List of *dcf.cashflows.payoffs.OptionCashFlowPayOff*.

class dcf.cashflows.contingent.**OptionStrategyCashflowList**(*payment_date_list*, *call_amount_list=1.0*, *call_strike_list=()*, *put_amount_list=1.0*, *put_strike_list=()*, *fixing_offset=None*, *pay_offset=None*, *origin=None*, *payoff_model=None*)

Bases: *ContingentCashFlowList*

list of option strategy cashflows

series of identical option strategies

Parameters

- **payment_date_list** – list of cashflow payment dates t_k
- **call_amount_list** – list of call option notional amounts N_i
- **call_strike_list** – list of call option strikes K_i
- **put_amount_list** – list of put option notional amounts N_j
- **put_strike_list** – list of put option strikes L_j
- **fixing_offset** – offset δ between underlying fixing date and cashflow end date
- **pay_offset** – offset ϵ between cashflow end date and payment date
- **origin** – origin of object, i.e. start date of the cashflow list as a product
- **payoff_model** – payoff model to derive the expected payoff

dcf.cashflows.contingent.OptionStrategyCashflowList object provides a list of *dcf.cashflows.payoffs.OptionStrategyCashFlowPayOff* X_k objects with payment date t_k.

Adjustetd by offset X_k has expiry date $T_k = t_k - \delta - \epsilon$ and for all k the same N_i, K_i, N_j, L_j are used.

class dcf.cashflows.contingent.**ContingentRateCashFlowList**(*payment_date_list*, *amount_list=1.0*, *origin=None*, *day_count=None*, *fixing_offset=None*, *pay_offset=None*, *fixed_rate=0.0*, *cap_strike=None*, *floor_strike=None*, *payoff_model=None*)

Bases: *ContingentCashFlowList*

list of cashflows by interest rate payments

list of contingend collared rate cashflows

Parameters

- **payment_date_list** – pay dates, assuming that pay dates agree with end dates of interest accrued period
- **amount_list** – notional amounts
- **origin** – start date of first interest accrued period
- **day_count** – day count convention
- **fixed_rate** – agreed fixed rate
- **forward_curve** –
- **fixing_offset** – time difference between interest rate fixing date and interest period payment date

- **pay_offset** – time difference between interest period end date and interest payment date
- **floor_strike** – lower interest rate boundary K
- **cap_strike** – upper interest rate boundary L
- **payoff_model** – option valuation model to derive the expected cashflow of option payoffs

Each object consists of a list of *dcf.cashflows.payoffs.ContingentRateCashFlowPayOff*, i.e. of collared payoff functions

$$X_i(f(T_i)) = [\max(K, \min(f(T_i), L)) + c]\ \tau(s, e)\ N$$

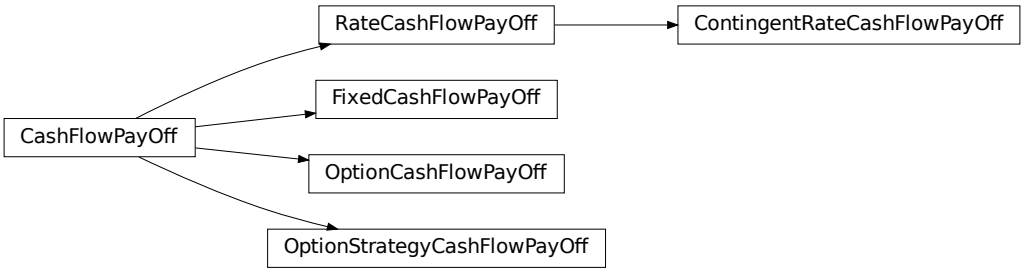
with, according to a payment date p_i, $p_i - \epsilon = e_i$, $e_i = s_{i+1}$ and $s_i - \delta = T_i$.

payoff_model

model to derive the expected cashflow of an option payoff

property fixed_rate

3.2.4 Contingent Cashflow PayOffs



class dcf.cashflows.payoffs.**CashFlowPayOff**

Bases: object

Cash flow payoff base class

details(_=*None*)

class dcf.cashflows.payoffs.**FixedCashFlowPayOff**(*amount=1.0*)

Bases: *CashFlowPayOff*

fixed cashflow payoff

Parameters
amount – notional amount N

A fixed cashflow payoff X is given directly by the notional amount N

Invoking $X()$ or $X(m)$ with a *dcf.models.optionpricing.OptionPayOffModel* object m as argument returns the actual expected cashflow payoff amount of X which is again just the notional amount N.

```
>>> from dcf import FixedCashFlowPayOff
>>> cf = FixedCashFlowPayOff(123.456)
>>> cf()
123.456
```

details(*_=None*)

class dcf.cashflows.payoffs.**RateCashFlowPayOff**(*start*, *end*, *amount=1.0*, *day_count=None*, *fixing_offset=None*, *fixed_rate=0.0*)

Bases: *CashFlowPayOff*

interest rate cashflow payoff

Parameters

- **start** – cashflow accrued period start date s
- **end** – cashflow accrued period end date e
- **amount** – notional amount N
- **day_count** – function to calculate accrued period year fraction τ
- **fixing_offset** – time difference between interest rate fixing date and interest period payment date δ
- **fixed_rate** – agreed fixed rate c

A contigent interest rate cashflow payoff X is given for a float rate f at $T = s - \delta$

$$X(f(T)) = (f(T) + c)\,\tau(s, e)\,N$$

Invoking $X(m)$ with a *dcf.models.optionpricing.OptionPayOffModel* object m as argument returns the actual expected cashflow payoff amount of X.

```
>>> from dcf import RateCashFlowPayOff, CashRateCurve
```

```
>>> cf = RateCashFlowPayOff(start=1.25, end=1.5, amount=1.0, fixed_rate=0.005)
>>> f = CashRateCurve(domain=[0.0, 1.0, 2.0], data=[-0.005, 0.00, 0.001],␣
↪forward_tenor=0.25)
```

```
>>> cf()
0.00125
>>> cf(f)
0.0013125
```

start

interest accrued period start date

end

interest accrued period end date

day_count

interest accrued period day count method for rate period calculation τ

fixing_offset

time difference between interest rate fixing date and interest period payment date

amount

cashflow notional amount

fixed_rate

agreed fixed rate c

details(*forward_curve=None*)

class dcf.cashflows.payoffs.**OptionCashFlowPayOff**(*expiry*, *amount=1.0*, *strike=None*, *is_put=False*)

Bases: *CashFlowPayOff*

European option payoff function

Parameters

- **expiry** – option exipry date T
- **amount** – option notional amount N
- **strike** – strike price K
- **is_put** – bool **True** for put options and **False** for call options (optional with default **False**)

An European call option $C_K(S(T))$ is the right to buy an agreed amount N of an asset with future price $S(T)$ at a future point in time T (the option exipry date) for a pre-agreed strike price K.

The call option payoff provides the expected profit from such transaction, i.e.

$$C_K(S(T)) = N \cdot E[\max(S(T) - K, 0)]$$

Resp. a put option $P_K(S(T))$ is the right to sell an asset at a pre-agreed strike price. Hence, the put option payoff provides the expected profit from such transaction, i.e.

$$P_K(S(T)) = N \cdot E[\max(K - S(T), 0)]$$

As the asset price $S(t)$ is unknown at time $t < T$, the estimation of $C_K(S(T))$ resp. $P_K(S(T))$ requires assumptions on the as randomness understood unkown behavior of S until T.

This is provided by **payoff_model** implementing *dcf.models.optionpricing.OptionPayOffModel* and is invoked by calling an *dcf.cashflows.payoffs.OptionCashFlowPayOff* object.

First, setup a classical log-normal *Black-Scholes* model.

```
>>> from dcf.models import LogNormalOptionPayOffModel
>>> from math import exp
>>> f = lambda t: 100.0 * exp(t * 0.05)  # spot price 100 and yield of 5%
>>> v = lambda v: 0.1  # flat volatility of 10%
>>> m = LogNormalOptionPayOffModel(valuation_date=0.0, forward_curve=f,␣
↪volatility_curve=v)
```

Then, build a call option payoff.

```
>>> from dcf import OptionCashFlowPayOff
>>> c = OptionCashFlowPayOff(expiry=0.25, strike=110.0)
>>> # get expected option payoff
>>> c(m)
0.10726740675017865
```

And a put option payoff.

```
>>> p = OptionCashFlowPayOff(expiry=0.25, strike=110.0, is_put=True)
>>> # get expected option payoff
>>> p(m)
8.849422252686733
```

details(*model=None*)

class dcf.cashflows.payoffs.**OptionStrategyCashFlowPayOff**(*expiry*, *call_amount_list=1.0*, *call_strike_list=()*, *put_amount_list=1.0*, *put_strike_list=()*)

Bases: *CashFlowPayOff*

option strategy, i.e. series of call and put options with single expiry

Parameters

- **expiry** – option exiptry date T
- **call_amount_list** – list of call option notional amounts N_i
- **call_strike_list** – list of call option strikes K_i
- **put_amount_list** – list of put option notional amounts N_j
- **put_strike_list** – list of put option strikes L_j

The option strategy payoff X is the sum of call and put payoffs

$$X(S(T)) = \sum_{i=1}^{m} N_i \cdot C_{K_i}(S(T)) + \sum_{j=1}^{n} N_j \cdot P_{L_j}(S(T))$$

see more on options strategies

First, setup a classical log-normal *Black-Scholes* model.

```
>>> from dcf.models import LogNormalOptionPayOffModel
>>> from math import exp
>>> #
>>> f = lambda t: 100.0 * exp(t * 0.05)  # spot price 100 and yield of 5%
>>> v = lambda v: 0.1  # flat volatility of 10%
>>> m = LogNormalOptionPayOffModel(valuation_date=0.0, forward_curve=f,␣
↪volatility_curve=v)
```

Then, setup a *butterlfy* payoff and evaluate it.

```
>>> from dcf import OptionStrategyCashFlowPayOff
>>> call_amount_list = 1., -2., 1.
>>> call_strike_list = 100, 110, 120
>>> s = OptionStrategyCashFlowPayOff(expiry=1., call_amount_list=call_amount_
↪list, call_strike_list=call_strike_list)
>>> s(m)
3.06924777745399
```

details(*model=None*)

class dcf.cashflows.payoffs.**ContingentRateCashFlowPayOff**(*start*, *end*, *amount=1.0*, *day_count=None*, *fixing_offset=None*, *fixed_rate=0.0*, *floor_strike=None*, *cap_strike=None*)

Bases: *RateCashFlowPayOff*

contigent but collared interest rate cashflow payoff

Parameters

- **start** – cashflow accrued period start date s
- **end** – cashflow accrued period end date e
- **amount** – notional amount N
- **day_count** – function to calculate accrued period year fraction τ
- **fixing_offset** – time difference between interest rate fixing date and interest period payment date δ
- **fixed_rate** – agreed fixed rate c
- **floor_strike** – lower interest rate boundary K
- **cap_strike** – upper interest rate boundary L

A collared interest rate cashflow payoff X is given for a float rate f at $T = s - \delta$

$$X(f(T)) = [\max(K, \min(f(T), L)) + c]\ \tau(s, e)\ N$$

The foorlet $(\max(K, \dots))$ or resp. the caplet condition $(\min(\dots, L))$ will be ignored if K is or resp. L is **None**.

Invoking $X(m)$ with a *dcf.models.optionpricing.OptionPayOffModel* object m as argument returns the actual expected cashflow payoff amount of X.

```
>>> from dcf import ContingentRateCashFlowPayOff, CashRateCurve
>>> from dcf.models import NormalOptionPayOffModel, IntrinsicOptionPayOffModel
```

evaluate just the fixed rate cashflow

```
>>> cf = ContingentRateCashFlowPayOff(start=1.25, end=1.5, amount=1.0, fixed_
↪rate=0.005, floor_strike=0.002)
>>> cf()
0.00125
```

evaluate the fixed rate and float forward rate cashflow

```
>>> f = CashRateCurve(domain=[0.0, 1.0, 2.0], data=[-0.005, 0.00, 0.001],␣
↪forward_tenor=0.25)
>>> cf(f)
0.0013125
```

evaluate the fixed rate and float forward rate cashflow plus intrisic option payoff

```
>>> i = IntrinsicOptionPayOffModel(valuation_date=0.0, forward_curve=f)
>>> cf(i)
0.00175
```

evaluate the fixed rate and float forward rate cashflow plus *Bachelier* model payoff

```
>>> m = NormalOptionPayOffModel(valuation_date=0.0, forward_curve=f, volatility_
↪curve=(lambda *_: 0.005))
>>> cf(m)
0.0021158872175425702
```

floor_strike

floor strike rate

cap_strike

cap strike rate

details(*model=None*)

3.2.5 Contingent Cashflow Models

dcf.models.intrinsic.IntrinsicOptionPayOffModel	intrisic option pricing formula
dcf.models.bachelier.NormalOptionPayOffModel	Bachelier option pricing formula
dcf.models.black76.LogNormalOptionPayOffModel	Black 76 option pricing formula
dcf.models.displaced.DisplacedLogNormalOptionPayOffModel	displaced Black 76 option pricing formula

class dcf.models.optionpricing.**OptionPricingFormula**

Bases: `object`

abstract base class for option pricing formulas

A *dcf.models.optionpricing.OptionPricingFormula* f serves as a kind of interface template to enhance *dcf.models.optionpricing.OptionPayOffModel* by a new model.

To do so, f should at least implement a method

- **__call__(time, strike, forward, volatility)**

to provide the expected payoff of an Europen call option. Alternativly, it could implement the same signatur for a private

- **_call_price(time, strike, forward, volatility)**

method. These and all follwing method are only related to call options since put options will be derivend by the use of put-call parity.

Moreover, the **volatility** argument should be understood as a general input of model parameters which ar in case of classical option pricing formulas like *dcf.models.black76.LogNormalOptionPayOffModel* the volatility.

To provide non-numerical derivatives implement

- **_call_delta(time, strike, forward, volatility)**

for delta Δ_f, the first derivative along the underlying

- **_call_gamma(time, strike, forward, volatility)**

for gamma Γ_f, the second derivative along the underlying

- **_call_vega(time, strike, forward, volatility)**

for vega $\mathcal{V}_f$, the first derivative along the volatility parameters

- **_call_theta(time, strike, forward, volatility)**

for theta Θ_f, the first derivative along the time parameter **time**

classmethod from_function(*func*, *name=None*)

create new type (class) of an OptionPricingFormula

Parameters
func – function serving as **__call__** method as in *dcf.models.optionpricing.OptionPricingFormula*

Returns
subclass of *dcf.models.optionpricing.OptionPricingFormula*

class dcf.models.optionpricing.**OptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPricingFormula*

base option payoff model to derive expected payoff cashflows

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

DELTA_SHIFT = 0.0001
finite difference to calculate numerical delta sensitivities

DELTA_SCALE = 0.0001
factor to express numerical delta sensitivities usually in a value of a basis point (bpv)

Let δ be the **DELTA_SHIFT** and ϵ be the **DELTA_SCALE** and f a forward F sensitive function such that

$$f' = \frac{df}{dF} \approx \Delta_f(F) = \frac{f(F+\delta) - f(x)}{\delta/\epsilon}.$$

VEGA_SHIFT = 0.01
finite difference to calculate numerical vega sensitivities

VEGA_SCALE = 0.01
factor to express numerical vega sensitivities

Let δ be the **VEGA_SHIFT** and ϵ be the **VEGA_SCALE** and f a volatility ν sensitive function such that

$$f'_\nu = \frac{df}{d\nu} \approx \mathcal{V}_f(\nu) = \frac{f(\nu+\delta) - f(\nu)}{\delta/\epsilon}.$$

THETA_SHIFT = 0.0027378507871321013
finite difference to calculate numerical theta sensitivities usually one day (1/365.25)

THETA_SCALE = 0.0027378507871321013
factor to express numerical theta sensitivities usually one day (1/365.25)

Let δ be the **THETA_SHIFT** and ϵ be the **THETA_SCALE** and f a time $\tau(t,T)$ sensitive function with valuation date t and option maturity date T such that

$$\dot{f} = \frac{df}{dt} \approx \Theta_f(t) = \frac{f(\tau(t,T)+\delta) - f(\tau(t,T))}{\delta/\epsilon}.$$

valuation_date

date of option valuation t

forward_curve

curve for deriving forward values $F(t)$

volatility_curve

parameter curve of option pricing formulas $\nu(t)$

day_count

day count function to calculate year fraction between dates τ

details(*date*, *strike=None*)

model parameter details

Parameters

- **date** – option expiry date (also fixing date)
- **strike** – option strike value (optional; default **None**, i.e. *at-the-money*)

Returns

dict()

get_call_value(*date*, *strike=None*)

value of a call option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$C_K(F(T)) = E[\max(F(T) - K, 0)]$

get_put_value(*date*, *strike=None*)

value of a put option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$P_K(F(T)) = E[\max(K - F(T), 0)]$

Note $P_K(F(T))$ is derived by put-call parity:

$$P_K(F(T)) = K - F(T) + C_K(F(T))$$

get_call_delta(*date*, *strike=None*)

delta sensitivity of a call option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\Delta_{C_K(F)} = \frac{d}{dF} C_K(F)$

$\Delta_{C_K(F)}$ is the first derivative of $C_K(F)$ in unterlying direction F.

get_put_delta(*date*, *strike=None*)

delta sensitivity of a put option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\Delta_{P_K(F)} = \frac{d}{dF} P_K(F)$

$\Delta_{P_K(F)}$ is the first derivative of $P_K(F)$ in unterlying direction F and is derived by put-call parity, too:

$$\Gamma_{P_K(F)} = \Delta_{C_K(F)} - 1$$

Note, here 1 is actualy scaled by *dcf.models.optionpricing.OptionPayOffModel.DELTA_SCALE*.

get_call_gamma(*date*, *strike=None*)

gamma sensitivity of a call option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\Gamma_{C_K(F)} = \frac{d^2}{dF^2} C_K(F)$

$\Gamma_{C_K(F)}$ is the second derivative of $C_K(F)$ in unterlying direction F.

get_put_gamma(*date*, *strike=None*)

gamma sensitivity of a put option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\Gamma_{P_K(F)} = \frac{d^2}{dF^2} P_K(F)$

$\Gamma_{P_K(F)}$ is the second derivative of $P_K(F)$ in unterlying direction F and is derived by put-call parity, too:

$$\Gamma_{P_K(F)} = \Gamma_{C_K(F)}$$

get_call_vega(*date*, *strike=None*)

vega sensitivity of a call option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\mathcal{V}_{C_K(F)} = \frac{d}{dv} C_K(F)$

$\mathcal{V}_{C_K(F)}$ is the first derivative of $C_K(F)$ in volatility parameter direction v.

get_put_vega(*date*, *strike=None*)

vega sensitivity of a put option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\mathcal{V}_{P_K(F)} = \frac{d}{dv} P_K(F)$

$\mathcal{V}_{P_K(F)}$ is the first derivative of $P_K(F)$ in volatility parameter direction v and is derived by put-call parity, too:

$$\mathcal{V}_{P_K(F)} = VC_K(F)$$

get_call_theta(*date*, *strike=None*)

time sensitivity of a call option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\Theta_{C_K(F)} = \frac{d}{dt} C_K(F)$

$\Theta_{C_K(F)}$ is the first derivative of $C_K(F)$ in time parameter direction, i.e. valuation date t.

get_put_theta(*date*, *strike=None*)

time sensitivity of a put option

Parameters

- **date** – expiry date T
- **strike** – option strike price K of underlying F

Returns

$\Theta_{P_K(F)} = \frac{d}{dt} P_K(F)$

$\Theta_{P_K(F)}$ is the first derivative of $P_K(F)$ in time parameter direction, i.e. valuation date t.

class dcf.models.optionpricing.**BinaryOptionPayOffModel**(*pricing_formula*, *strike_shift=None*, *valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*)

Bases: *OptionPayOffModel*

biniary option payoff model (derived by finite differences)

Parameters

- **pricing_formula** – option pricing formula; eithter *dcf.models.optionpricing.OptionPricingFormula* or *dcf.models.optionpricing.OptionPayOffModel*
- **strike_shift** – finite difference to calculate binary option payoff as a call spread (optional: default taken from *dcf.models.optionpricing.BinaryOptionPayOffModel.STRIKE_SHIFT*)
- **valuation_date** – date of option valuation t (optional: default taken from **pricing_formula**)
- **forward_curve** – curve for deriving forward values (optional: default taken from **pricing_formula**)

- **volatility_curve** – parameter curve of option pricing formulas (optional: default taken from **pricing_formula**)
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date (optional: default taken from **pricing_formula**)

Let δ be the **STRIKE_SHIFT** and f a option payoff with strike K such that the binary payoff is given as

$$f' = \frac{df}{dK} \approx \frac{f(K+\delta/2) - f(K-\delta/2)}{\delta}.$$

STRIKE_SHIFT = 0.0001

finite difference to calculate binary option payoff as a call spread

class dcf.models.intrinsic.**IntrinsicOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPayOffModel*

intrisic option pricing formula

implemented for call options (see more on intrisic option values)

Let F be the current forward value. Let K be the option strike value, τ the time to matruity, i.e. the option expitry date.

Then

- call price:

$$\max(F-K, 0)$$

- call delta:

$$0 \text{ if } F < K \text{ else } 1$$

- call gamma:

$$0$$

- call vega:

$$0$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.intrinsic.**BinaryIntrinsicOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPayOffModel*

intrisic option pricing formula for binary call options (see also *dcf.models.intrinsic.IntrinsicOptionPayOffModel*)

Let F be the current forward value. Let K be the option strike value, τ the time to matruity, i.e. the option expitry date.

Then

- call price:

$$0 \text{ if } F < K \text{ else } 1$$

- call delta:

$$0$$

- call gamma:

$$0$$

- call vega:

$$0$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.bachelier.**NormalOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPayOffModel*

Bachelier option pricing formula

implemented for call options (see more on Bacheliers model)

Let f be a normaly distributed random variable with expectation $F = E[f]$, the current forward value and Φ the standard normal cummulative distribution function s.th. $\phi = \Phi'$ is its density function.

Let K be the option strike value, τ the time to matruity, i.e. the option expitry date, and σ the volatility parameter, i.e. the standard deviation of f. Moreover, let

$$d = \frac{F - K}{\sigma \cdot \sqrt{\tau}}$$

Then

- call price:

$$(F - K) \cdot \Phi(d) + \sigma \cdot \sqrt{\tau} \cdot \phi(d)$$

- call delta:

$$\Phi(d)$$

- call gamma:

$$\frac{\phi(d)}{\sigma \cdot \sqrt{\tau}}$$

- call vega:

$$\sqrt{\tau} \cdot \phi(d)$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.bachelier.**BinaryNormalOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPayOffModel*

Bachelier option pricing formula for binary calls (see also *dcf.models.bachelier.NormalOptionPayOffModel*)

Let f be a normaly distributed random variable with expectation $F = E[f]$, the current forward value and Φ the standard normal cummulative distribution function s.th. $\phi = \Phi'$ is its density function.

Let K be the option strike value, τ the time to matruity, i.e. the option expitry date, and σ the volatility parameter, i.e. the stanard deviation of f. Moreover, let

$$d = \frac{F - K}{\sigma \cdot \sqrt{\tau}}$$

Then

- call price:

$$\Phi(d)$$

- call delta:

$$\frac{\phi(d)}{\sigma \cdot \sqrt{\tau}}$$

- call gamma:

$$d \cdot \frac{\phi(d)}{\sigma^2 \cdot \tau}$$

- call vega:

$$\sqrt{\tau} \cdot \phi(d)$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.black76.**LogNormalOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPayOffModel*

Black 76 option pricing formula

implemented for call options (see more on Black 76 model which is closly related to the Black-Scholes model)

Let f be a log-normaly distributed random variable with expectation $F = E[f]$, the current forward value and Φ the standard normal cummulative distribution function s.th. $\phi = \Phi'$ is its density function.

Let K be the option strike value, τ the time to maturity, i.e. the option expiry date, and σ the volatility parameter, i.e. the standard deviation of $\log(f)$. Moreover, let

$$d = \frac{\log(F/K) + (\sigma^2 \cdot \tau)/2}{\sigma \cdot \sqrt{\tau}}$$

Then

- call price:

$$F \cdot \Phi(d) - K \cdot \Phi(d - \sigma \cdot \sqrt{\tau})$$

- call delta:

$$\Phi(d)$$

- call gamma:

$$\frac{\phi(d)}{F \cdot \sigma \cdot \sqrt{\tau}}$$

- call vega:

$$F \cdot \sqrt{\tau} \cdot \phi(d)$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.black76.**BinaryLogNormalOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*)

Bases: *OptionPayOffModel*

Black 76 option pricing formula for binary calls (see also *dcf.models.black76.LogNormalOptionPayOffModel*)

Let f be a log-normaly distributed random variable with expectation $F = E[f]$, the current forward value and Φ the standard normal cummulative distribution function s.th. $\phi = \Phi'$ is its density function.

Let K be the option strike value, τ the time to maturity, i.e. the option expiry date, and σ the volatility parameter, i.e. the standard deviation of $\log(f)$. Moreover, let

$$d = \frac{\log(F/K) + (\sigma^2 \cdot \tau)/2}{\sigma \cdot \sqrt{\tau}}$$

Then

- call price:

$$\Phi(d)$$

- call delta:

$$\frac{\phi(d - \sigma \cdot \sqrt{\tau})}{\sigma \cdot \sqrt{\tau}}$$

- call gamma:

$$d \cdot \frac{\phi(d)}{\sigma^2 \cdot \tau}$$

- call vega:

$$(d/\sigma - \sqrt{\tau}) \cdot \phi(d)$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas

- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.displaced.**DisplacedLogNormalOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*, *displacement=0.0*)

Bases: *LogNormalOptionPayOffModel*

displaced Black 76 option pricing formula

implemented for call options (see also *dcf.models.black76.LogNormalOptionPayOffModel*)

The displaced Black 76 is adopted to handel moderate negative underlying forward prices or rates f, e.g. as see for interest rates in the past. To do so, rather than f a shifted or displaced version $f + \alpha$ is assumed to be log-normaly distributed for some negative value of α.

Hence, let $f + \alpha$ be a log-normaly distributed random variable with expectation $F + \alpha = E[f + \alpha]$, where $F = E[f]$ is the current forward value and Φ the standard normal cummulative distribution function s.th. $\phi = \Phi'$ is its density function.

Let K be the option strike value, τ the time to maturity, i.e. the option expiry date, and σ the volatility parameter, i.e. the standard deviation of $\log(f + \alpha)$. Moreover, let

$$d = \frac{\log((F + \alpha)/(K + \alpha)) + (\sigma^2 \cdot \tau)/2}{\sigma \cdot \sqrt{\tau}}$$

Then

- call price:

$$(F + \alpha) \cdot \Phi(d) - (K + \alpha) \cdot \Phi(d - \sigma \cdot \sqrt{\tau})$$

- call delta:

$$\Phi(d)$$

- call gamma:

$$\frac{\phi(d)}{(F + \alpha) \cdot \sigma \cdot \sqrt{\tau}}$$

- call vega:

$$(F + \alpha) \cdot \sqrt{\tau} \cdot \phi(d)$$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date

- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

class dcf.models.displaced.**BinaryDisplacedLogNormalOptionPayOffModel**(*valuation_date=None*, *forward_curve=None*, *volatility_curve=None*, *day_count=None*, *bump_greeks=None*, *displacement=0.0*)

Bases: *BinaryLogNormalOptionPayOffModel*

displaced Black 76 option pricing formula for binary calls (see also *dcf.models.displaced.DisplacedLogNormalOptionPayOffModel*)

Uses *dcf.models.black76.BinaryLogNormalOptionPayOffModel* formulas with

- displaced forward $F + \alpha$

and

- displaced strike $K + \alpha$

option payoff model

Parameters

- **valuation_date** – date of option valuation t
- **forward_curve** – curve for deriving forward values
- **volatility_curve** – parameter curve of option pricing formulas
- **day_count** – day count function to calculate year fraction between dates, e.g. option expiry and valueation date
- **bump_greeks** – **bool** - if **True** Greeks, i.e. sensitivities/derivatives, are derived numericaly. If **False** analytics functions are used, if given. See also *dcf.models.optionpricing.OptionPricingFormula*. (optional; default is **False**)

3.3 Valuation Routines

3.3.1 Present Value

dcf.pricer.**get_present_value**(*cashflow_list*, *discount_curve*, *valuation_date=None*)

calculates the present value by discounting cashflows

Parameters

- **cashflow_list** – list of cashflows
- **discount_curve** – discount factors are obtained from this curve
- **valuation_date** – date to discount to (optional; default: **discount_curve.origin**)

Returns

float - as the sum of all discounted future cashflows

Let $cf_1 \dots cf_n$ be the list of cashflows with payment dates $t_1, \dots, t_n$.

Moreover, let t be the valuation date and $T = \{t_i \mid t \leq t_i\}$.

Then the present value is given as

$$v(t) = \sum_{t_i \in T} df(t, t_i) \cdot cf_i$$

with $df(t, t_i)$, the discount factor discounting form t_i to t.

Note, **get_present_value** includes cashflows at valuation date. Therefor it represents a *start-of-day* valuation than a *end-of-day* valuation.

```
>>> from dcf import CashFlowList, ZeroRateCurve, get_present_value
>>> cfs = CashFlowList([0, 1, 2, 3],[100, 100, 100, 100])
>>> curve = ZeroRateCurve([0], [0.05])
>>> valuation_date = 0
>>> sod = get_present_value(cfs, curve, valuation_date)
>>> sod
371.67748189617316
>>> eod = sod - cfs[valuation_date]
>>> eod
271.67748189617316
```

3.3.2 Yield To Maturity

dcf.pricer.**get_yield_to_maturity**(*cashflow_list*, *valuation_date=None*, *present_value=0.0*, *precision=1e-07*, *bounds=(- 0.1, 0.2)*, ***kwargs*)

yield-to-maturity or effective interest rate

Parameters

- **cashflow_list** – list of cashflows
- **valuation_date** – date to discount to (optional; default: **cashflow_list.origin**)
- **present_value** – price to meet by discounting (optional; default: 0.0)
- **precision** – max distance of present value to par (optional: default is 1e-7)
- **bounds** – tuple of lower and upper bound of yield to maturity (optional: default is -0.1 and .2)
- **kwargs** – additional keyword used for constructing *dcf.curves.interestratecurve.ZeroRateCurve*

Returns

float - as flat interest rate to discount all future cashflows in order to meet given **present_value**

Let $cf_1 \dots cf_n$ be the list of cashflows with payment dates $t_1, \dots, t_n$.

Moreover, let t be the valuation date and $T = \{t_i \mid t \leq t_i\}$.

Then the yield-to-maturity is the interest rate y such that the **present_value** $\hat{v}$ is given as

$$\hat{v} = \sum_{t_i \in T} df(t, t_i) \cdot cf_i$$

with $df(t, t_i) = \exp(-y \cdot (t_i - t))$, the discount factor discounting form t_i to t.

Example

yield-to-matrurity of 5y fixed coupon bond

```
>>> from dcf import RateCashFlowList, FixedCashFlowList, CashFlowLegList
>>> from dcf import get_present_value, get_yield_to_maturity
>>> from dcf import ZeroRateCurve
```

```
>>> n, df = 1e6, ZeroRateCurve([0], [0.015])
>>> coupon_leg = RateCashFlowList([1,2,3,4,5], amount_list=n, origin=0, fixed_
↪rate=0.001)
>>> redemption_leg = FixedCashFlowList([5], amount_list=n)
>>> bond = CashFlowLegList((redemption_leg, coupon_leg))
```

bond with cashflow tables

```
>>> print(tabulate(coupon_leg.table, headers='firstrow'))
  cashflow    pay date    notional    start date    end date    year fraction  ␣
↪fixed rate
----------  ----------  ----------  ------------  ----------  ---------------  --
↪----------
      1000           1       1e+06             0           1                1  ␣
↪     0.001
      1000           2       1e+06             1           2                1  ␣
↪     0.001
      1000           3       1e+06             2           3                1  ␣
↪     0.001
      1000           4       1e+06             3           4                1  ␣
↪     0.001
      1000           5       1e+06             4           5                1  ␣
↪     0.001
```

```
>>> print(tabulate(redemption_leg.table, headers='firstrow'))
  cashflow    pay date
----------  ----------
     1e+06           5
```

get yield-to-maturity at par (gives coupon rate)

```
>>> ytm = get_yield_to_maturity(bond, valuation_date=0, present_value=n)
>>> round(ytm, 6)
0.001
```

get current yield-to-maturity as given by 1.5% risk free rate (gives risk free rate)

```
>>> pv = get_present_value(bond, df, valuation_date=0)
>>> pv
932524.5493034503
>>>
>>> ytm = get_yield_to_maturity(bond, valuation_date=0, present_value=pv)
>>> round(ytm, 6)
0.015
```

3.3.3 Fair Rate

dcf.pricer.**get_fair_rate**(*cashflow_list*, *discount_curve*, *valuation_date=None*, *present_value=0.0*, *precision=1e-07*, *bounds=(- 0.1, 0.2)*)

coupon rate to meet given value

Parameters

- **cashflow_list** – list of cashflows
- **discount_curve** – discount factors are obtained from this curve
- **valuation_date** – date to discount to
- **present_value** – price to meet by discounting
- **precision** – max distance of present value to par (optional: default is 1e-7)
- **bounds** – tuple of lower and upper bound of fair rate (optional: default is -0.1 and .2)

Returns

float - the fair coupon rate as **fixed_rate** of a *dcf.cashflows.cashflow.RateCashFlowList*

Let $cf_i(c) = N_i \cdot \tau(s_i, e_i) \cdot (c + f(d_i))$ be the i-th cashflow in the **cashflow_list**.

Here, the fair rate is the fixed_rate $c = \hat{c}$ such that the **present_value** $\hat{v}$ is given as

$$\hat{v} = \sum_{t_i \in T} df(t, t_i) \cdot cf_i(\hat{c})$$

with $df(t, t_i)$, the discount factor discounting form t_i to t.

Note, **get_fair_rate** requires the **cashflow_list** to have an attribute **fixed_rate** which is perturbed to find the solution for $\hat{c}$.

Example

```
>>> from dcf import RateCashFlowList, FixedCashFlowList, CashFlowLegList
>>> from dcf import get_present_value, get_fair_rate
>>> from dcf import ZeroRateCurve
```

setup 5y coupon bond

```
>>> n, df = 1e6, ZeroRateCurve([0], [0.015])
>>> coupon_leg = RateCashFlowList([1,2,3,4,5], amount_list=n, origin=0, fixed_
↪rate=0.001)
>>> redemption_leg = FixedCashFlowList([5], amount_list=n)
>>> bond = CashFlowLegList((redemption_leg, coupon_leg))
```

find fair rate to give par bond

```
>>> pv = get_present_value(redemption_leg, df)
>>> fair_rate = get_fair_rate(coupon_leg, df, present_value=n-pv)
>>> fair_rate
0.015113064615715653
```

check it's a par bond (pv=notional)

```
>>> coupon_leg.fixed_rate = fair_rate
>>> pv = get_present_value(bond, df)
>>> round(pv, 6)
1000000.0
```

3.3.4 Interest Accrued

dcf.pricer.**get_interest_accrued**(*cashflow_list*, *valuation_date*)

calculates interest accrued for rate cashflows

Parameters

- **cashflow_list** – requires a *day_count* property
- **valuation_date** – calculation date

Returns

float - proportion of interest in current interest period

Let t be the valuation date and s, e start resp. end date of current rate period, i.e. $s \leq t < e$.

Let τ be the day count function to calculate year fractions.

Finally, let cf be the next interest rate cashflow.

The accrued interest until t is given as

$$cf_{accrued} = cf \cdot \frac{\tau(s,t)}{\tau(s,e)}.$$

Note, this function takes even expected payoffs of options incl. caplets and floorlets into account which probably should be excluded.

Example

```
>>> from dcf import RateCashFlowList, FixedCashFlowList, CashFlowLegList
>>> from dcf import get_interest_accrued
```

setup 5y coupon bond

```
>>> n = 1e6
>>> coupon_leg = RateCashFlowList([1,2,3,4,5], amount_list=n, origin=0, fixed_
↪rate=0.001)
>>> redemption_leg = FixedCashFlowList([5], amount_list=n)
>>> bond = CashFlowLegList((redemption_leg, coupon_leg))
```

bond with cashflow tables

```
>>> print(tabulate(coupon_leg.table, headers='firstrow'))
  cashflow    pay date    notional    start date    end date    year fraction  ␣
↪fixed rate
----------  ----------  ----------  ------------  ----------  ---------------  --
↪----------
      1000           1       1e+06             0           1                1  ␣
↪     0.001
      1000           2       1e+06             1           2                1  ␣
↪     0.001
      1000           3       1e+06             2           3                1  ␣
↪     0.001
```

(continues on next page)

(continued from previous page)

```
      1000            4        1e+06              3            4                1  ␣
↪      0.001
      1000            5        1e+06              4            5                1  ␣
↪      0.001
```

```
>>> print(tabulate(redemption_leg.table, headers='firstrow'))
  cashflow    pay date
----------  ----------
     1e+06           5
```

calculate accrued interest

```
>>> get_interest_accrued(bond, valuation_date=3.25)
250.0
```

```
>>> get_interest_accrued(bond, valuation_date=3.5)
500.0
```

```
>>> # doesn't take fixed cashflows into account
>>> get_interest_accrued(bond, valuation_date=4.5)
500.0
```

3.3.5 Basis Point Value

dcf.pricer.**get_basis_point_value**(*cashflow_list*, *discount_curve*, *valuation_date=None*, *delta_curve=None*, *shift=0.0001*)

basis point value (bpv), i.e. value change by one interest rate shifted one basis point

Parameters

- **cashflow_list** – list of cashflows
- **discount_curve** – discount factors are obtained from this curve
- **valuation_date** – date to discount to
- **delta_curve** – curve (or list of curves) which will be shifted
- **shift** – shift size to derive bpv

Returns

float - basis point value (bpv)

Let $v(t, r)$ be the present value of the given **cashflow_list** depending on interest rate curve r which can be used as forward curve to estimate float rates or as zero rate curve to derive discount factors (or both).

Then, with **shift_size** s, the bpv is given as

$$\Delta(t) = 0.0001 \cdot \frac{v(t, r+s) - v(t, r)}{s}$$

Example

```
>>> from dcf import RateCashFlowList, FixedCashFlowList, CashFlowLegList
>>> from dcf import get_present_value, get_basis_point_value
>>> from dcf import ZeroRateCurve
```

setup 5y coupon bond

```
>>> n, df = 1e6, ZeroRateCurve([0], [0.015])
>>> coupon_leg = RateCashFlowList([1,2,3,4,5], amount_list=n, origin=0, fixed_
↪rate=0.001)
>>> redemption_leg = FixedCashFlowList([5], amount_list=n)
>>> bond = CashFlowLegList((redemption_leg, coupon_leg))
```

calculate bpv as bond delta

```
>>> bpv = get_basis_point_value(bond, df)
>>> bpv
-465.1755130274687
```

check by direct valuation

```
>>> pv = get_present_value(bond, df)
>>> df[0] += 0.0001
>>> shifted = get_present_value(bond, df)
>>> shifted-pv
-465.1755130274687
```

3.3.6 Bucketed Delta

dcf.pricer.**get_bucketed_delta**(*cashflow_list*, *discount_curve*, *valuation_date=None*, *delta_curve=None*, *delta_grid=None*, *shift=0.0001*)

list of bpv delta for partly shifted interest rate curve

Parameters

- **cashflow_list** – list of cashflows
- **discount_curve** – discount factors are obtained from this curve
- **valuation_date** – date to discount to (optional; default is **discount_curve.origin**)
- **delta_curve** – curve (or list of curves) which will be shifted (optional; default is **discount_curve**)
- **delta_grid** – grid dates to build partly shifts (optional; default is **delta_curve.domain**)
- **shift** – shift size to derive bpv (optional: default is a basis point i.e. *0.0001*)

Returns

list(float) - basis point value for each **delta_grid** point

Let $v(t, r)$ be the present value of the given **cashflow_list** depending on interest rate curve r which can be used as forward curve to estimate float rates or as zero rate curve to derive discount factors (or both).

Then, with **shift_size** s and shifting s_j,

$$\Delta_j(t) = 0.0001 \cdot \frac{v(t, r + s_j) - v(t, r)}{s}$$

and the full bucketed delta vector is $\big(\Delta_1(t), \Delta_2(t), \ldots, \Delta_{m-1}(t)\Delta_m(t)\big)$.

Overall the shifting $s_1, \dots s_n$ is a partition of the unity, i.e. $\sum_{j=1}^{m} s_j = s$.

Each s_j for $i = 2, \dots, m-1$ is a function of the form of an triangle, i.e. for a **delta_grid** $t_1, \dots, t_m$

$$s_j(t) = \begin{cases} 0 & \text{for } t < t_{j-1} \\ s \cdot \frac{t - t_{j-1}}{t_j - t_{j-1}} & \text{for } t_{j-1} \le t < t_j \\ s \cdot \frac{t_{j+1} - t}{t_{j+1} - t_j} & \text{for } t_j \le t < t_{j+1} \\ 0 & \text{for } t_{j+1} \le t \end{cases}$$

while

$$s_1(t) = \begin{cases} s & \text{for } t < t_1 \\ s \cdot \frac{t_2 - t}{t_2 - t_1} & \text{for } t_1 \le t < t_2 \\ 0 & \text{for } t_2 \le t \end{cases}$$

and

$$s_m(t) = \begin{cases} 0 & \text{for } t < t_{m-1} \\ s \cdot \frac{t - t_{m-1}}{t_m - t_{m-1}} & \text{for } t_{m-1} \le t < t_m \\ s & \text{for } t_m \le t \end{cases}$$

Example

same example as *dcf.pricer.get_basis_point_value()* but with buckets

```
>>> from dcf import RateCashFlowList, FixedCashFlowList, CashFlowLegList
>>> from dcf import get_present_value, get_bucketed_delta, get_basis_point_value
>>> from dcf import ZeroRateCurve
```

setup 5y coupon bond

```
>>> n, df = 1e6, ZeroRateCurve([0,1,2,3,4,5], [0.01, 0.011, 0.014, 0.012, 0.01,␣
↪0.013])
>>> coupon_leg = RateCashFlowList([1,2,3,4,5], amount_list=n, origin=0, fixed_
↪rate=0.001)
>>> redemption_leg = FixedCashFlowList([5], amount_list=n)
>>> bond = CashFlowLegList((redemption_leg, coupon_leg))
```

calculate bpv as bond delta

```
>>> bpv = get_bucketed_delta(bond, df)
>>> bpv
(0.0, -0.09890108276158571, -0.19445822690613568, -0.2893486835528165, -0.
↪38423892273567617, -468.88503439340275)
```

check by summing up (should give flat bpv)

```
>>> sum(bpv)
-469.85198130935896
>>> get_basis_point_value(bond, df)
-469.85198130935896
```

3.3.7 Curve Bootstrapping

dcf.pricer.**get_curve_fit**(*cashflow_list*, *discount_curve*, *valuation_date=None*, *fitting_curve=None*, *fitting_grid=None*, *present_value=0.0*, *precision=1e-07*, *bounds=(- 0.1, 0.2)*)

fit curve to cashflow_list prices (bootstrapping)

Parameters

- **cashflow_list** – list (!) of cashflow_list. products to match prices
- **discount_curve** – discount factors are obtained from this curve
- **valuation_date** – date to discount to
- **fitting_curve** – curve to fit to match prices (optional; default is **discount_curve**)
- **fitting_grid** – domain to fit prices to (optional; default **fitting_curve.domain**)
- **present_value** – list (!) of prices of products in **cashflow_list**, each one to be met by discounting (optional; default is list of 0.0)
- **precision** – max distance of present value to par (optional; default is 1e-7)
- **bounds** – tuple of lower and upper bound of fair rate (optional; default is -0.1 and .2)

Returns

tuple(float) **fitting_data** as curve values to build curve together with curve points from **fitting_grid**

Bootstrapping is a sequential approach to set curve values y_i in order to match present values of cashflow products X_j to given prices p_j.

This is done sequentially, i.e. for a consecutive sequence of curve points t_i, $i = 1 \dots n$.

Starting at $i = 1$ at curve point t_1 the value y_1 is varied by simple bracketing such that the present value

$$v_0(X_j) = p_j \text{ for } X_j \text{ maturing before or at } t_j.$$

Here, the **fitting_curve** can be the discount curve but also any forward curve or even a volatility curve.

Once y_1 is found the next dated t_2 in **fitting_grid** is handled. This is kept going on until all prices p_j and all points t_i match.

Note, in order to conclude successfully, the valuations $v_0(X_j)$ must be sensitive to changes of curve value y_i, i.e. at least one j must hold

$$\frac{d}{dy_i} v_0(X_j) \neq 0$$

with in the bracketing bounds of y_i as set by **bounds**.

Example

Yield curve calibration.

First, setup dates and schedule

```
>>> from businessdate import BusinessDate, BusinessSchedule
>>> today = BusinessDate(20161231)
>>> schedule = BusinessSchedule(today + '1y', today + '5y', '1y')
```

of products

```
>>> from dcf import RateCashFlowList, get_present_value
>>> cashflow_list = [RateCashFlowList([s for s in schedule if s <= d], 1e6,␣
↪origin=today, fixed_rate=0.01) for d in schedule]
```

and prices to match to.

```
>>> from dcf import ZeroRateCurve
>>> rates = [0.01, 0.009, 0.012, 0.014, 0.011]
>>> curve = ZeroRateCurve(schedule, rates)
>>> present_value = [get_present_value(cfs, curve, today) for cfs in cashflow_
↪list]
```

Then fit a plain curve

```
>>> from dcf import get_curve_fit
>>> target = ZeroRateCurve(schedule, [0.01, 0.01, 0.01, 0.01, 0.01])
>>> data = get_curve_fit(cashflow_list, target, today, fitting_curve=target,␣
↪present_value=present_value)
>>> [round(d, 6) for d in data]
[0.01, 0.009, 0.012, 0.014, 0.011]
```

Example

Option implied volatility calibration.

First, setup dates and schedule

```
>>> from businessdate import BusinessDate, BusinessSchedule
>>> today = BusinessDate(20161231)
>>> expiry = today + '3m'
```

curves

```
>>> from dcf import ZeroRateCurve, ForwardCurve, TerminalVolatilityCurve
>>> c = ZeroRateCurve([today], [0.05])   # risk free rate of 5%
>>> f = ForwardCurve([today], [100.0], yield_curve=c)  # spot price 100 and␣
↪yield of 5%
>>> v = TerminalVolatilityCurve([today], [0.1])  # flat volatility of 10%
```

and model with parameters

```
>>> from dcf.models import LogNormalOptionPayOffModel
>>> m = LogNormalOptionPayOffModel(valuation_date=today, forward_curve=f,␣
↪volatility_curve=v)
```

of call option products

```
>>> from dcf import OptionCashflowList, get_present_value
>>> cashflow_list = OptionCashflowList([expiry], strike_list=110., origin=today,␣
↪payoff_model=m)
>>> cashflow_list[expiry]
0.1025451675720177
>>> get_present_value(cashflow_list, curve, today)
0.1022928005931384
```

and fit volatility by

```
>>> from dcf import get_curve_fit
>>> pv = 0.25
>>> data = get_curve_fit([cashflow_list], curve, today, fitting_curve=v, fitting_
↪grid=[expiry], present_value=[pv])
>>> data
(0.12207840979099276,)
```

check result

```
>>> v[expiry] = data[0]
>>> pv = get_present_value(cashflow_list, curve, today)
>>> round(pv, 6)
0.25
```

3.4 Fundamentals

3.4.1 Interpolation

class dcf.interpolation.**base_interpolation**(*x_list=[]*, *y_list=[]*)

Bases: object

Basic class to interpolate given data.

interpolation class

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

classmethod from_dict(*xy_dict*)

create an interpolation instance object from **dict**

Parameters
xy_dict – dictionary with sorted keys serving as **x_list** and values as **y_list**

Returns
interpolation object

class dcf.interpolation.**flat**(*y=0.0*)

Bases: *base_interpolation*

flat or constant interpolation

Parameters
y – constant return value $\hat{y}$

A *dcf.interpolation.flat* object is a function f returning a constant value $\hat{y}$.

$$f(x) = \hat{y} \text{ const.}$$

for all x.

```
>>> from dcf.interpolation import flat
>>> c = flat(1.1)
>>> c(0)
1.1
>>> c(2.1)
1.1
```

class dcf.interpolation.**default_value_interpolation**(*x_list=[]*, *y_list=[]*, *default_value=None*)

Bases: *base_interpolation*

default value interpolation

Parameters

- **x_list** – points $x_1 \dots x_n$

- **y_list** – values $y_1 \dots y_n$
- **default_value** – default value d

A *dcf.interpolation.default_value_interpolation* object is a function f returning at x the value y_i with i to be the first matching index such that $x = x_i$

$$f(x) = y_i \text{ for } x = x_i$$

and d if no matching x_i is found.

```
>>> from dcf.interpolation import default_value_interpolation
>>> c = default_value_interpolation([1,2,3,1], [1,2,3,4], default_value=42)
>>> c(1)
1.0
>>> c(2)
2.0
>>> c(4)
42
```

class dcf.interpolation.**no**(*x_list=[]*, *y_list=[]*)

Bases: *default_value_interpolation*

no interpolation at all

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.no* object is a function f returning at x the value y_i with i to be the first matching index such that $x = x_i$

$$f(x) = y_i \text{ for } x = x_i \text{ else None}$$

```
>>> from dcf.interpolation import no
>>> c = no([1,2,3,1], [1,2,3,4])
>>> c(1)
1.0
>>> c(2)
2.0
>>> c(4)
```

class dcf.interpolation.**zero**(*x_list=[]*, *y_list=[]*)

Bases: *default_value_interpolation*

interpolation by filling with zeros between points

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.zero* object is a function f returning at x the value y_i if $x = x_i$ else zero. with i to be the first matching index such that

$$f(x) = y_i \text{ for } x = x_i \text{ else } 0$$

```
>>> from dcf.interpolation import zero
>>> c = zero([1,2,3,1], [1,2,3,4])
>>> c(1)
1.0
>>> c(1.1)
0.0
>>> c(2)
2.0
>>> c(4)
0.0
```

class dcf.interpolation.**left**(*x_list=[]*, *y_list=[]*)

Bases: *base_interpolation*

left interpolation

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.left* object is a function f returning at x the last given value y_i reading from left to right, i.e. with i to be the matching index such that

$$f(x) = y_i \text{ for } x_i \leq x < x_{i+1}$$

or y_1 if $x < x_1$.

```
>>> from dcf.interpolation import left
>>> c = left([1,3], [1,2])
>>> c(0)
1.0
>>> c(1)
1.0
>>> c(2)
1.0
>>> c(4)
2.0
```

class dcf.interpolation.**constant**(*x_list=[]*, *y_list=[]*)

Bases: *left*

constant interpolation

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

Same as *dcf.interpolation.left*.

class dcf.interpolation.**right**(*x_list=[]*, *y_list=[]*)

Bases: *base_interpolation*

right interpolation

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.right* object is a function f returning at x the last given value y_i reading from right to left, i.e. with i to be the matching index such that

$$f(x) = y_i \text{ for } x_i < x \leq x_{i+1}$$

or y_n if $x_n < x$.

```
>>> from dcf.interpolation import right
>>> c = right([1,3], [1,2])
>>> c(0)
1.0
>>> c(1)
1.0
>>> c(2)
2.0
>>> c(4)
2.0
```

class dcf.interpolation.**nearest**(*x_list=[]*, *y_list=[]*)

Bases: *base_interpolation*

nearest interpolation

Parameters

- **x_list** – points $x_1 \ldots x_n$
- **y_list** – values $y_1 \ldots y_n$

A *dcf.interpolation.nearest* object is a function f returning at x the given value y_i of the nearest x_i from both left and right, i.e. with i to be the matching index such that

$$f(x) = y_i \text{ for } \mid x_i - x \mid = \min_j \mid x_j - x \mid$$

```
>>> from dcf.interpolation import nearest
>>> c = nearest([1,2,3], [1,2,3])
>>> c(0)
1.0
>>> c(1)
1.0
>>> c(1.5)
1.0
>>> c(1.51)
2.0
>>> c(2)
2.0
>>> c(4)
3.0
```

class dcf.interpolation.**linear**(*x_list=[]*, *y_list=[]*)

Bases: *base_interpolation*

linear interpolation

Parameters

- **x_list** – points $x_1 \ldots x_n$
- **y_list** – values $y_1 \ldots y_n$

A *dcf.interpolation.linear* object is a function f returning at x the linear interpolated value of y_i and y_{i+1} when $x_i \leq x < x_{i+1}$, i.e.

$$f(x) = (y_{i+1} - y_i) \cdot \frac{x - x_i}{x_{i+1} - x_i}$$

```
>>> from dcf.interpolation import linear
>>> c = linear([1,2,3], [2,3,4])
>>> c(0)
1.0
>>> c(1)
2.0
>>> c(1.5)
2.5
>>> c(1.51)
2.51
>>> c(2)
3.0
>>> c(4)
5.0
```

class dcf.interpolation.**loglinear**(*x_list=[]*, *y_list=[]*)

Bases: *linear*

log-linear interpolation

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.loglinear* object is a function f returning at x the value $\exp(y)$ of the linear interpolated value y of $\log(y_i)$ and $\log(y_{i+1})$ when $x_i \leq x < x_{i+1}$, i.e.

$$f(x) = \exp\left((\log(y_{i+1}) - \log(y_i)) \cdot \frac{x - x_i}{x_{i+1} - x_i}\right)$$

```
>>> from math import log, exp
>>> from dcf.interpolation import loglinear
>>> c = loglinear([1,2,3], [exp(2),exp(3),exp(4)])
>>> log(c(0))
1.0
>>> log(c(1))
2.0
>>> log(c(1.5))
2.5
>>> log(c(1.51))
2.51
>>> log(c(2))
3.0
>>> log(c(4))
5.0
```

Note: **loglinear** requires strictly positive values $0 < y_1 \dots y_n$.

class dcf.interpolation.**loglinearrate**(*x_list=[]*, *y_list=[]*)

Bases: *linear*

log-linear interpolation by annual rates

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.loglinearrate* object is a function f returning at x the value $\exp(x \cdot y)$ of the linear interpolated value y of $\log(\frac{y_i}{x_i})$ and $\log(\frac{y_{i+1}}{x_{i+1}})$ when $x_i \leq x < x_{i+1}$, i.e.

$$f(x) = \exp\left(x \cdot (\log(\frac{y_{i+1}}{x_{i+1}}) - \log(\frac{y_i}{x_i})) \cdot \frac{x - x_i}{x_{i+1} - x_i}\right)$$

```
>>> from math import log, exp
>>> from dcf.interpolation import loglinear
>>> c = loglinear([1,2,3], [exp(1*2),exp(2*3),exp(2*4)])
>>> log(c(0))
-2.0
>>> log(c(1))
2.0
>>> log(c(1.5))
4.0
>>> log(c(1.51))
4.04
>>> log(c(2))
6.0
>>> log(c(4))
10.0
```

Note: **loglinear** requires strictly positive values $0 < y_1 \dots y_n$.

class dcf.interpolation.**logconstantrate**(*x_list=[]*, *y_list=[]*)

Bases: *constant*

log-constant interpolation by annual rates

Parameters

- **x_list** – points $x_1 \dots x_n$
- **y_list** – values $y_1 \dots y_n$

A *dcf.interpolation.logconstantrate* object is a function f returning at x the value $\exp(x \cdot y)$ of the constant interpolated value y of $\log(\frac{y_i}{x_i})$ when $x_i \leq x < x_{i+1}$, i.e.

$$f(x) = \exp\left(x \cdot \log(\frac{y_i}{x_i})\right)$$

```
>>> from math import log, exp
>>> from dcf.interpolation import logconstantrate
>>> c = logconstantrate([1,2,3], [exp(1*2),exp(2*3),exp(2*4)])
>>> log(c(1))
2.0
>>> log(c(1.5))
3.0
>>> log(c(1.51))
3.02
>>> log(c(2))
6.0
>>> log(c(3))
8.0
```

Note: **logconstantrate** requires strictly positive values $0 < y_1 \dots y_n$.

class dcf.interpolation.**interpolation_scheme**(*domain*, *data*, *interpolation=None*)

Bases: object

class to build piecewise interpolation function

Parameters

- **domain** (*list(float)*) – points $x_1 \dots x_n$
- **data** (*list(float)*) – values $y_1 \dots y_n$
- **interpolation** (*function*) – interpolation function on **x_list** (optional) or triple of (**left**, **mid**, **right**) interpolation functions with
  - **left** for $x < x_1$ (as default **right** is used)
  - **mid** for $x_1 \leq x \leq x_n$ (as default *dcf.interpolation.linear* is used)
  - **right** for $x > x_n$ (as default *dcf.interpolation.constant* is used)

Curve object to build function

$$f(x) = y$$

using piecewise various interpolation functions.

class dcf.interpolation.**constant_linear_constant**(*domain*, *data*, *interpolation=None*)

Bases: *interpolation_scheme*

class to build piecewise interpolation function

Parameters

- **domain** (*list(float)*) – points $x_1 \dots x_n$
- **data** (*list(float)*) – values $y_1 \dots y_n$
- **interpolation** (*function*) – interpolation function on **x_list** (optional) or triple of (**left**, **mid**, **right**) interpolation functions with
  - **left** for $x < x_1$ (as default **right** is used)
  - **mid** for $x_1 \leq x \leq x_n$ (as default *dcf.interpolation.linear* is used)
  - **right** for $x > x_n$ (as default *dcf.interpolation.constant* is used)

Curve object to build function

$$f(x) = y$$

using piecewise various interpolation functions.

dcf.interpolation.**linear_scheme**

alias of *constant_linear_constant*

dcf.interpolation.**logconstant_loglinear_logconstant**

alias of constant_loglinear_constant

dcf.interpolation.**log_linear_scheme**

alias of constant_loglinear_constant

class dcf.interpolation.**logconstantrate_loglinearrate_logconstantrate**(*domain*, *data*, *interpolation=None*)

Bases: *interpolation_scheme*

class to build piecewise interpolation function

Parameters

- **domain** (*list(float)*) – points $x_1 \dots x_n$

- **data** (*list(float)*) – values $y_1 \dots y_n$
- **interpolation** (*function*) – interpolation function on **x_list** (optional) or triple of (**left**, **mid**, **right**) interpolation functions with
  - **left** for $x < x_1$ (as default **right** is used)
  - **mid** for $x_1 \leq x \leq x_n$ (as default *dcf.interpolation.linear* is used)
  - **right** for $x > x_n$ (as default *dcf.interpolation.constant* is used)

Curve object to build function

$$f(x) = y$$

using piecewise various interpolation functions.

dcf.interpolation.**log_linear_rate_scheme**

alias of *logconstantrate_loglinearrate_logconstantrate*

class dcf.interpolation.**zero_linear_constant**(*domain*, *data*, *interpolation=None*)

Bases: *interpolation_scheme*

class to build piecewise interpolation function

Parameters

- **domain** (*list(float)*) – points $x_1 \dots x_n$
- **data** (*list(float)*) – values $y_1 \dots y_n$
- **interpolation** (*function*) – interpolation function on **x_list** (optional) or triple of (**left**, **mid**, **right**) interpolation functions with
  - **left** for $x < x_1$ (as default **right** is used)
  - **mid** for $x_1 \leq x \leq x_n$ (as default *dcf.interpolation.linear* is used)
  - **right** for $x > x_n$ (as default *dcf.interpolation.constant* is used)

Curve object to build function

$$f(x) = y$$

using piecewise various interpolation functions.

dcf.interpolation.**zero_linear_scheme**

alias of *zero_linear_constant*

3.4.2 Compounding

dcf.compounding.**simple_compounding**(*rate_value*, *maturity_value*)

simple compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ

Returns

$\frac{1}{1+r \cdot \tau}$

dcf.compounding.**simple_rate**(*df*, *period_fraction*)

interest rate from simple compounded dicount factor

Parameters

- **df** – discount factor df
- **period_fraction** – interest rate period τ

Returns

$\frac{1}{df-1} \cdot \frac{1}{\tau}$

dcf.compounding.**continuous_compounding**(*rate_value*, *maturity_value*)

continuous compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ

Returns

$\exp(-r \cdot \tau)$

dcf.compounding.**continuous_rate**(*df*, *period_fraction*)

interest rate from continuous compounded dicount factor

Parameters

- **df** – discount factor df
- **period_fraction** – interest rate period τ

Returns

$-\log(df) \cdot \frac{1}{\tau}$

dcf.compounding.**periodic_compounding**(*rate_value*, *maturity_value*, *period_value*)

periodicly compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ
- **period_value** – number of interest rate periods m

Returns

$(1 + \frac{r}{m})^{-\tau \cdot m}$

dcf.compounding.**periodic_rate**(*df*, *period_fraction*, *frequency*)

interest rate from continuous compounded dicount factor

Parameters

- **df** – discount factor df
- **period_fraction** – interest rate period τ
- **frequency** – number of interest rate periods m

Returns

$(df^{-\frac{1}{\tau \cdot m}} - 1) \cdot m$

dcf.compounding.**annually_compounding**(*rate_value*, *maturity_value*)

annually compounded discount factor

Parameters

- **rate_value** – interest rate r

- **maturity_value** – loan maturity τ

Returns
$(1+r)^{-\tau}$

dcf.compounding.**semi_compounding**(*rate_value*, *maturity_value*)

semi compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ

Returns
$(1+\frac{r}{2})^{-\tau \cdot 2}$

dcf.compounding.**quarterly_compounding**(*rate_value*, *maturity_value*)

quarterly compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ

Returns
$(1+\frac{r}{4})^{-\tau \cdot 4}$

dcf.compounding.**monthly_compounding**(*rate_value*, *maturity_value*)

monthly compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ

Returns
$(1+\frac{r}{12})^{-\tau \cdot 12}$

dcf.compounding.**daily_compounding**(*rate_value*, *maturity_value*)

daily compounded discount factor

Parameters

- **rate_value** – interest rate r
- **maturity_value** – loan maturity τ

Returns
$(1+\frac{r}{365})^{-\tau \cdot 365}$

3.4.3 DayCount

dcf.daycount.**day_count**(*start*, *end*)

default day count function for rate period calculation

Parameters

- **start** – period start date t_s
- **end** – period end date t_e

Returns
year fraction $\tau(t_s, t_e)$ from **start** to **end** as a float

this default **day_count** function calculates the number of days between t_s and t_e expressed as a fraction of a year, i.e.

$$\tau(t_s, t_e) = \frac{t_e - t_s}{365.25}$$

as an average year has nearly 365.25 days.

Since different date packages have differnet concepts to derive the number of days between two dates, **day_count** tries to adopt at least some of them. As there are:

- dates given already as year fractions as a float so $\tau(t_s, t_e) = t_e - t_s$.
- datetime the native Python package, so $\delta = t_e - t_s$ is a **timedelta** object with attribute **days** which is used.
- businessdate a specialised package for banking business calendar and time period calculations, so the **BusinessDate** object **start** has a method **start.diff_in_days** which is used.

CHAPTER

FOUR

RELEASES

These changes are listed in decreasing version number order.

4.1 Release 0.7

Release date was Tuesday, 31 May 2022

- added *dcf.curves.curve.Curve.kwargs* to clone and persist object
- added *dcf.curves.curve.ForwardCurve* for asset forwards like stocks or commodities
- added *dcf.curves.fx.FxForwardCurve* for fx forwards rates
- added *dcf.cashflows.cashflow.CashFlowList.kwargs* to clone and persist object
- added *dcf.cashflows.contingent* for option pricing
- added various standard option pricing formulas `dcf.models` incl. digital or binary versions like
  - *Bachelier* as *dcf.models.bachelier.NormalOptionPayOffModel*
  - *Black-Scholes* resp. *Black76* as *dcf.models.black76.LogNormalOptionPayOffModel*
  - *displaced Black76* as *dcf.models.displaced.DisplacedLogNormalOptionPayOffModel*
  - as well as an intrinsic version *dcf.models.intrinsic.IntrinsicOptionPayOffModel*
- modified *dcf.pricer.get_present_value()* to word with *dcf.models.optionpricing.OptionPayOffModel*
- removed *dcf.pricer.get_fair_rate()* alias *get_par_rate()*
- removed submodules *dcf.data*
- added submodules *dcf.cashflows.payoffs*
- added pricer routine *dcf.pricer.get_curve_fit()*
- added *dcf.curves.curve.RateCurve.spread*

4.2 Release 0.6

Release date was Friday, 14 January 2022

- added `dcf.cashflows.cashflow.FixedCashFlowList.table` and `dcf.cashflows.cashflow.RateCashFlowList.table`
- **added new module *dcf.daycount* and updated *dcf.daycount.day_count()***
  to default to year fractions in case of simple float inputs
- added *dcf.pricer.get_bucketed_delta()*
- added submodules *dcf.data*, *dcf.cashflows.products* and *dcf.curves.plot* (under construction)

4.3 Release 0.5

Release date was November 22, 2021

- added *dcf.pricer.get_basis_point_value()*

4.4 Release 0.4

Release date was October 11, 2020

- dropping support for python 2 incl. 2.7
- new casting concept for curves, old *curve_instance.cast(TypeToCastTo)* is replaced by *TypeToCastTo(curve_instance)*
- restructuring cashflow lists, see *dcf.cashflows.cashflow*
- adding payment plans, see *dcf.plans*
- adding pricing functions, e.g. *dcf.pricer.get_present_value()*, *dcf.pricer.get_yield_to_maturity()*, *get_par_rate()*, ...
- more docs
- more tests

4.5 Release 0.3

Release date was September 18, 2019

- migration to python 3.4, 3.5, 3.6 and 3.7
- automated code review
- more docs
- supporting third party (e.g.) interpolation
- adding travis ci
- update for auxilium tools
- replaced *assert stmt* by *if not stmt: raise AssertionError()* (bandit recommendation)

4.6 Release 0.2

Release date was March 3, 2017

4.7 Release 0.1

Release date was December 31, 2016

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

E

F

G

Q

R

S

T

V

Y

Z